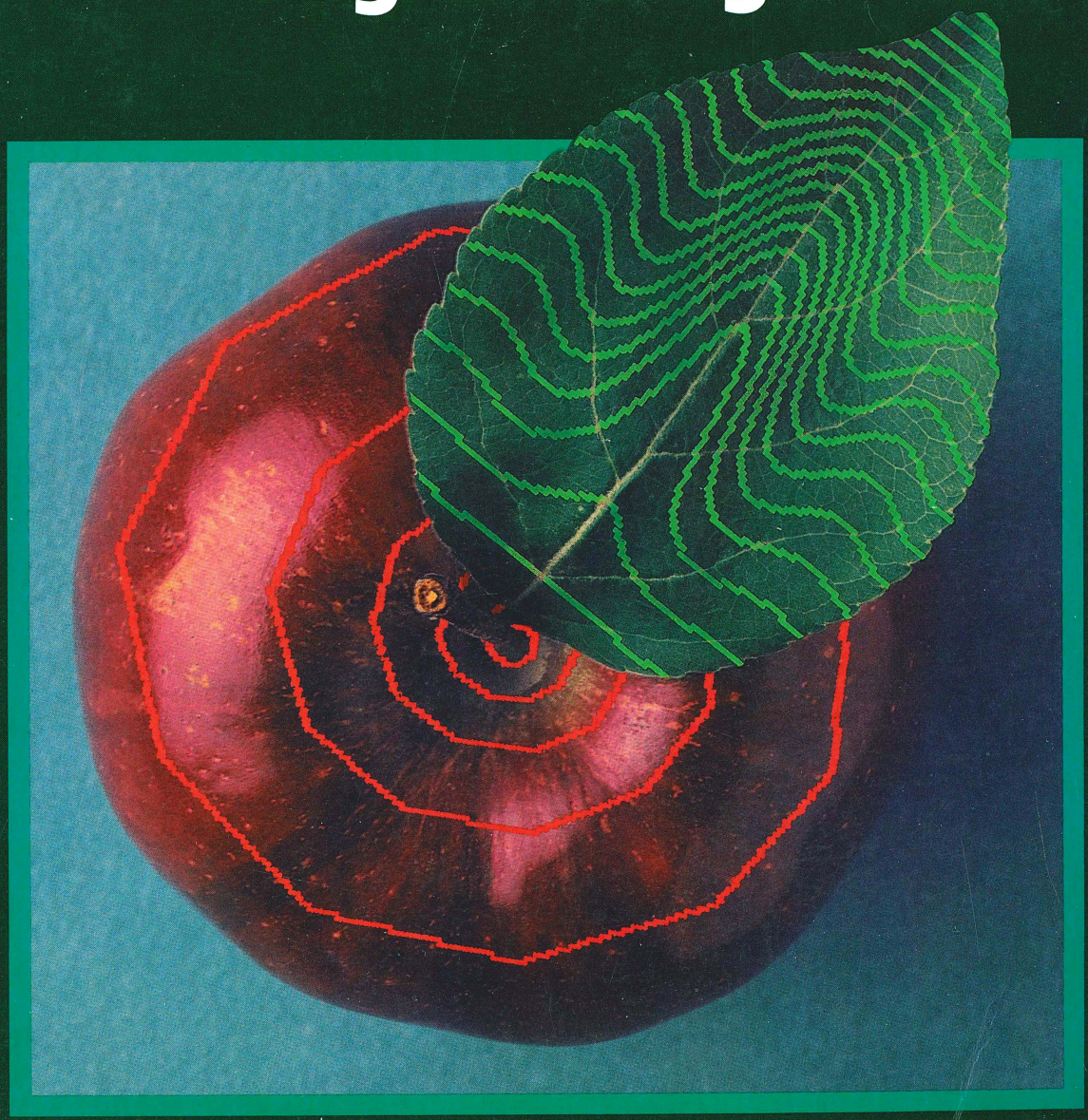
22342

SAMS

The Waite Group

Apple® Logo Programming Primer



Donald Martin Stephen Prata Marijane Axtel Paulsen

Apple® Logo Programming Primer

The Waite Group, Inc. is a San Rafael, California-based producer of high-quality books on personal computing. Acknowledged as a leader in the industry, the Waite Group has written and produced over thirty titles, including such best sellers as *Unix Primer Plus, Computer Graphics Primer, CP/M Primer, CP/M Bible* and *Soul of CP/M.* Internationally known and award-winning, Waite Group books are distributed world-wide and have been repackaged with the products of such major companies as Epson, Wang, Xerox, Tandy-Radio Shack, NCR, and Exxon Office Systems. Mr. Waite, president of the Waite Group, has been involved in the computer industry since 1976 when he bought his first Apple I computer from Steven Jobs.

Donald Martin has served as chairman of the Physics, Astronomy and Energy Science Department at the College of Marin in Kentfield, CA. He received his B.A. from the University of California, Berkeley and his M.A. from San Jose State University. He has long been interested in the problems that students have in developing their reasoning and critical thinking skills. Recently, this interest has led him to the Logo language, a course he now teaches at the college. Mr. Martin is coauthor of *Unix Primer Plus, C Primer Plus,* and *88 Apple Logo Programs.* His hobbies include reading, running, and traveling with his wife, Kay, and his family.

Marijane Axtel Paulsen is the chief instructional officer at the Santa Clarita Community College in Valencia, California. She received her B.S. degree from Arizona State University, her M.S. degree from Whittier College, and her Ph.D. from the University of Southern California. The former dean of science and technology at Indian Valley Colleges in Marin, California, her primary interests include teacher training and computer literacy. She and her husband, Paul, enjoy travel and cross country skiing.

Stephen Prata is a professor of physics and astronomy at the College of Marin in Kentfield, California. He received his B.S. from the California Institute of Technology and his Ph.D. from the University of California, Berkeley. His association with computers began with the computer modeling of star clusters. He is currently involved with teaching UNIX at the College of Marin. Mr. Prata is coauthor of *UNIX Primer Plus* and *C Primer Plus.* He and his wife, Kathleen, live in San Rafael, CA. Their interests include travel, table tennis, and photography.

APPLE® LOGO PROGRAMMING PRIMER

Featuring Top-Down Structured Programming

by

DONALD MARTIN
MARIJANE PAULSEN
STEPHEN PRATA

Howard W. Sams & Co., Inc.
4300 WEST 62ND ST. INDIANAPOLIS, INDIANA 46268 USA

Copyright ©1984 by The Waite Group, Inc.

FIRST EDITION
FIRST PRINTING—1984

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. While every precaution has been taken in the preparation of this book, the publisher assumes no responsibility for errors or omissions. Neither is any liability assumed for damages resulting from the use of the information contained herein.

International Standard Book Number: 0-672-22342-2
Library of Congress Catalog Card Number: 84-50735

Edited by *Patricia Perry*
Illustrated by *Ron Troxell*

Printed in the United States of America.

PREFACE

This is the first book designed to teach the revolutionary Logo programming language with a special emphasis on "structured programming." Logo is being widely hailed as the language destined to play a crucial role in introducing programming to first time computer users of all ages.

Logo's marvelous "turtle graphics" introduce key programming ideas quickly, easily and with powerful visual feedback. And Logo's untyped variables and extensibility naturally guide new users into the techniques of top-down structured program design.

This book emphasizes top-down program design by using the exclusive "box chart" method to plan, change, and debug Logo programs. These box charts also provide a way of explaining Logo's most powerful programming tool: recursion. In addition, the box charts encourage the beginning programmer to create "utilities" to improve the quality of program writing.

This book, and its companion, "88 Apple Logo Programs," are the only books that take special care to provide you with these seven proven learning aids:

*special focus on recursion and outputs
*formal, consistent definition of primitives
*box-chart approach to structured programming
*generous use of utilities to build complex programs
*clear, concise, diagrammatic explanation of Logo syntax
*how to write "humanized" programs
*stimulating programming projects

Within these pages you'll discover how easy it is to learn the complete Logo language and you'll be able to use Logo to plan and write useful and fascinating programs that do what you want.

DONALD MARTIN
STEPHEN PRATA
MARIJANE PAULSEN

We are extremely pleased to be able to offer *Logo Programming Primer* to the computing community. While turtle graphics is very easy for beginners, the full Logo language needs careful and thoughtful explanation. This book meets all our requirements. We believe that its many features, especially its explanation of Logo's syntax and use of box charts, will make it the standard Logo book, a classic primer in the Waite Group tradition.

MITCHELL WAITE
President of the WAITE GROUP

ACKNOWLEDGEMENTS

This book, like its companions, *Logo Primer for the IBM PC and PCjr, 88 Apple Logo Programs,* and *88 IBM Logo Programs,* has roots tunneling deeply in many directions. First of all we would like to thank the creators of Logo, the researchers and scientists—primarily at MIT—who have created and nurtured this magnificent language.

We would like to offer special thanks to Dr. Lois Flynn, who brought Brian Harvey to San Francisco State University in the summer of 1981 and who made it possible for Brian to bring Logo to a few enthusiastic teachers. Just as important, Brian provided a version of Logo to run under UNIX at the College Of Marin in Kentfield, California, planting the seed that flowered into this book.

Thanks also to Jon Foreman, who keeps UNIX and Logo running smoothly at the College Of Marin and Indian Valley Colleges and special thanks to Bob Petersen, Bernd Enders, Fred Schmitt, Dan Putterman, Nancy Zimfirescu, and Dick Rodgers, who keep everything else running smoothly, too.

Many, many students contributed ideas, enthusiasm and positive feedback during the germination period of the past two years. Their contributions are reflected throughout the book. Specifically, special thanks to Martin Parlan, Toby Rein, Jason Schwager, Larry Jacobson, Jim Karas, Robert Riddel, Mike Maxwell, Tony Fardella, Ray Gruenig, Frank Adamson, Laurie McDaniel, and Cal Astrin.

Special thanks to Judith Griesgraber, Gordon Bainbridge, Ted Gilchrist and Steven Cross for their early contributions to the book.

Many thanks to Bob Johnson for his help with the illustrations and fantastic cartoons.

The good people at Howard W. Sams can always be counted on to provide strong, generous support.

Likewise, Michael Pardee and Kim House of the Waite Group were helpful in many ways.

Finally, we'd like to publicly thank our family and friends for their many sacrifices and loving support at all times.

If any misconceptions or errors are found in this book, please do not fault our friends and associates; it is our doing.

CONTENTS

INTRODUCTION 13
Why This Book—Why Learn Programming—Why Learn Logo—What Is Logo—How to Use This Book.

CHAPTER 1

GETTING STARTED 19
1.0 Introduction—1.1 Hardware Overview: System Requirements for Logo—1.2 The Keyboard—1.3 Loading Logo—1.4 Using Logo Commands: An Introduction to Turtle Graphics—1.5 On Line Editing—1.6 Summary—1.7 Creating Procedures—1.8 Creating Procedures from Procedures—1.9 Summary—1.10 Summary of Commands—1.11 Review Questions and Exercises.

CHAPTER 2

"TURTLEVERSE": THE TURTLE'S UNIVERSE 47
2.0 Introduction—2.1 Command Level—2.2 Screen Modes in Logo—2.3 Turtle Boundaries—2.4 Adding Color to the Turtle's Tracks—2.5 Textscreen—2.6 Summary—2.7 Review Questions and Exercises.

CHAPTER 3

THE STRUCTURE OF LOGO 75
3.0 Introduction—3.1 An Overview of the Logo Language—3.2 Variables as Inputs to Procedures—3.3 Giving Formal Definitions to Primitives—3.4 Summary—3.5 Review Questions and Exercises.

CHAPTER 4

CREATING AND MANAGING YOUR RESOURCES 99
4.0 Introduction—4.1 Initializing a Disk to Save Work—4.2 Graphics Utilities—4.3 Logo's Workspace—4.4 Handling Your Disks—4.5 Packaging Procedures and Burying Them—4.6 Obtaining a Hard Copy—4.7 Summary—4.8 Review Questions and Exercises.

CHAPTER 5

DEVELOPING GOOD PROGRAMMING TECHNIQUES 123

5.0 Introduction—5.1 Three Stages of Programming Skill in Logo—5.2 Programming and Problem Solving—5.3 Using the Box Chart Method—5.4 Programming Projects Using the Top-Down Approach—5.5 Program Control—5.6 Another Top-Down Planning Example: The Paycheck—5.7 Summary—5.8 Review Questions and Exercises.

CHAPTER 6

RECURSION 147

6.0 Introduction—6.1 Simple Recursion: Four Examples—6.2 Recursion Looks Easy, It Isn't: Complex Recursion—6.3 Recursion Analogies—6.4 Earning Interest—6.5 Stopping Recursion with an Input—6.6 Summary—6.7 Review Questions and Exercises.

CHAPTER 7

NUMBERS, WORDS AND LISTS 175

7.0 Introduction—7.1 How Do We Identify Numbers, Words and Lists—7.2 Taking Words and Lists Apart—7.3 Summary—7.4 Review Questions and Exercises.

CHAPTER 8

BUILDING WORDS AND LISTS 197

8.0 Introduction—8.1 The Primitive WORD—8.2 The Primitives SENTENCE and LIST—8.3 The Primitives FPUT and LPUT—8.4 Other Word and List Handling Primitives—8.5 Sample Program: Haiku Poetry—8.6 Summary—8.7 Review Questions and Exercises.

CHAPTER 9

VARIABLES: INPUTS, OUTPUTS AND THE MAKE PRIMITIVE 215

9.0 Introduction—9.1 Inputs—9.2 The MAKE Primitive—9.3 Interactive Programs: Using MAKE with READLIST—9.4 Comparing Inputs with MAKE—9.5 Creating Variables with OUTPUT—9.6 Summary—9.7 Review Questions and Exercises.

CHAPTER 10

LOGO AND MATHEMATICS 245

10.0 Introduction—10.1 Numbers—10.2 Arithmetic—10.3 Geometry—10.4 Trigonometry—10.5 Algebra—10.6 Math Utilities in Statistics—10.7 Summary—10.8 Review Questions.

CHAPTER 11

MAKING DECISIONS WITH CONDITIONALS 275

11.0 Introduction—11.1 IF: Simple Form with Two Inputs—11.2 IF: Complete Form with Three Inputs—11.3 The Primitives AND, OR and Not—11.4 The Primitives TEST, IFTRUE and IFFALSE—11.5 Predicates—11.6 The Utility GETB.NUMBER: Fetching a Number—11.7 STEER: Fly Your Turtle with GET.KEY—11.8 Make Your Own Predicates—11.9 Summary—11.10 Review Questions and Exercises.

CHAPTER 12

HUMANIZING YOUR PROGRAMS 301

12.0 Introduction—12.1 What's an Interactive Program?—12.2 Error Trapping—12.3 Two New Input Utilities: GETB.YN And GETC.NUMBER—12.4 Sample Program: Math Quiz—12.5 Programming Aids—12.6 Screen Formatting Utilities—12.7 Adding Formatting to QUIZ—12.8 Summary—12.9 Review Questions and Exercises.

CHAPTER 13

TWO PROGRAMS: MAKEGRAPH AND STORE.DATA 327

13.0 Introduction—13.1 MAKEGRAPH: The Program Backbone—13.2 GET.INFO: Getting Numbers for the Graph—13.3 SHOW.DATA and GO.AHEAD?—13.4 SETUP.GRAPH: Setting Up the Graph—13.5 PLOT.DATA: Putting Points on the Screen—13.6 Trying It Out—13.7 Graphing from a File—13.8 STORE.DATA: Saving a List on Disk—13.9 INIT: Starting Out—13.10 GET.DATA: Data Input—13.11 SAVE.DATA: Saving the List—13.12 An Example—13.13 Summary—13.14 Review Questions and Exercises.

CHAPTER 14

HANGMAN 361

14.0 Introduction—14.1 A Hangman Overview—14.2 The BEGIN Procedure—14.3 The CHOOSEWORD Procedure—14.4 The SETSCENE Procedure—14.5 The DASHMAKE Procedure—14.6 The GETGUESS and PROCESS Procedures—14.7 The FINE Procedure—14.8 The OOPS Procedure—14.9 The Hangman Procedure—14.10 Closing Comments—14.11 Review Questions and Exercises.

CHAPTER 15

THE STATES AND CAPITALS GAME: PROPERTY LISTS 387

15.0 Introduction—15.1 Property List Primitives—15.2 States and Capitals—15.3 Extensions to the Game—15.4 Property Lists and Tables—15.5 Property Lists and Arrays—15.6 Property Lists and Data Bases—15.7 Summary—15.8 Review Questions and Exercises.

APPENDIX A

SUMMARY OF LOGO COMMANDS 409

APPENDIX B

ASCII CHART 429

APPENDIX C

MANAGING THE APPLE WORKSPACE 437

APPENDIX D

LOGO ERROR MESSAGES 439

APPENDIX E

APPLE LOGO STARTUP 443

APPENDIX F

SUMMARY OF LOGO FEATURES 447

INDEX 449

Note

Throughout this book, reversed letters (e.g., RETURN) designate keystrokes. That is, when this designation is used, it means the reversed letters are a single keystroke on the keyboard (or keys that are pressed simultaneously to produce a desired result) instead of being input as individual characters. This same designation is also used for individual letters which are pressed to initiate a desired action in the program.

INTRODUCTION

Why This Book?

Today, we are in the midst of an information explosion propelled by tremendous advances in computer hardware and software. Just as computer hardware, the "nuts and bolts" of a computer system, has been decreasing in cost while increasing in performance, so too has computer software become easier to use and more powerful.

Logo is one example of this evolving software. Logo is designed to give beginners control over sophisticated computing resources that can be used as tools for learning, playing and exploring. It is possible with Logo to write very simple programs that do very complex things, especially using "turtle-graphics."

In addition, Logo has most of the features of a good programming language, so by learning Logo one also learns good programming habits. In fact, a popular microcomputer journal, *Byte* magazine, recommended in its August 1982 issue that Logo replace BASIC as the first language for beginning microcomputer users. This is beginning to happen. In its January 23, 1984 issue, the weekly computer news magazine, *Infoworld,* stated that Logo is "unquestionably poised to replace BASIC as a language for first-time users." Before we explain why this is so, let's look at a more basic question.

Why Learn Programming?

Everyone should ask himself or herself the questions, "Why should I learn a computer language anyway? Is it really important? What will it do for me?" Without going into all of the issues in what has become a national debate, we'll give you two reasons for learning to program.

First, writing programs gives you more control over what is mankind's most useful "tool," the computer.

Second, writing programs exercises the mind. Millions of people around the globe spend enormous amounts of time and energy working towards physical fitness. How about "thinking fitness"? "Problem-solving fitness"?

Shouldn't we exercise the brain, too? We think so. And many researchers think that programming is a very effective way to do it. The process of planning, writing and testing programs provides a form of mental gymnastics that exercises and conditions our neural headquarters.

Why Learn Logo?

Why learn to program in Logo? Why not BASIC? Or Pascal or Fortran or COBOL? This question can easily raise hackles among proponents of each language. We will try not to enter the debate here, but simply state three reasons why we think you should choose Logo.

1. Logo is the easiest language for beginners to use.
2. Logo is the most fun to begin with.
3. Logo encourages good programming techniques.

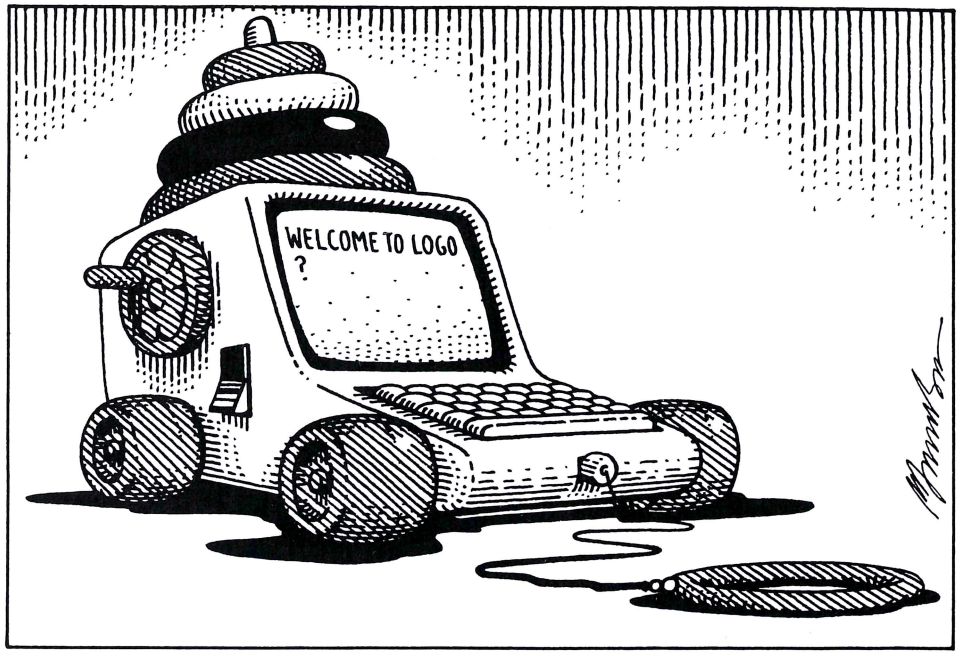
Let's see how Logo does this.

What Is Logo?

Logo has been described as being both a computer language and a philosophy of learning. The major theme of Logo's philosophy is to learn by doing, to learn by trial and error, to learn by exploring and discovering. Errors do not reflect failure, but "bugs" that can be analyzed and fixed.

The language itself was designed to reflect this philosophy by being easy to use yet powerful enough for sophisticated applications. Logo achieves these two seemingly opposing goals by incorporating several major design characteristics.

The most important Logo feature is its ability to respond to new user-created words or programs in the same way Logo responds to built-in words. Every computer language has certain "reserved" words that the computer knows and responds to. In Logo, these key words are called "primitives,"



and include such commands as, MAKE, FIRST, AND, IF, REPEAT, FORWARD and RIGHT. For example, the command

```
FORWARD 50
```

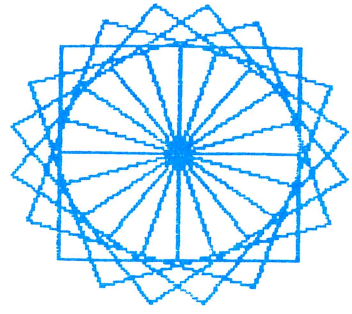
when given to the Logo language will cause a marvelous little fellow, called a "turtle," to move forward on the screen 50 "steps." No, there really isn't a four-legged reptile ambling across your screen; rather the "turtle" is an invention, a creation of Dr. Seymour Papert (one of the originators of Logo) which helps us visualize the results of writing programs. We control the turtle. We tell him or her (your choice) where to go and the turtle leaves tracks on the screen—sort of like the trails left by snails in the early morning sunshine. For example, the command

```
RIGHT 90
```

will turn the turtle 90 degrees.

We can use commands like these to form a program in Logo. For example, in chapter 1, we'll use these two commands to draw a picture of a box.

We can run this program, called a "procedure" in Logo, just by typing its name, the same as we run a Logo primitive. What is really wonderful about Logo is that we can now use our newly defined procedure—let's call it BOX—in a new program that we call TURNBOX. TURNBOX can use BOX to draw a box, then we instruct the turtle to turn right a bit, draw another box, and so on repeating as many times as we like.



Thus, we can build up Logo's basic vocabulary of words to "teach" the turtle new commands. Logo puts us in charge. We give the orders. The turtle marches to our instructions providing magnificent graphics which are highly motivating and personally satisfying.

What is more important is that this "modular" structure of Logo encourages us to build up pictures (and programs) from smaller pieces or "subprocedures."

Then, as we learn more of Logo and begin to take on larger programming projects, we do just the opposite. We divide large programming tasks up into more manageable parts. This is called "top-down" program design. It is one of the major topics of this book. Top-down program design reinforces in ourselves a very powerful problem-solving strategy. It is one of the qualities of Logo that has attracted so much interest from educators.

In addition to modularity, Logo has other important characteristics that contribute to its and to our success. These are summarized in Appendix F.

How to Use This Book

It is helpful to think of this book as consisting of three major parts, each requiring a slightly different approach. Briefly, here are the three parts and our suggestions for approaching them.

Part I Chapters 1–6

The emphasis here is on turtle graphics and Logo's structure.

The best approach is to "explore and discover."

If you've never used turtle graphics before, it is important that you spend sufficient time learning to write and edit procedures from memory. The tear-out Logo Startup Card in the back of this book is handy for use as a quick reference.

We suggest that you try at least two programming projects from Chapter 5 and that you fully explore recursion in Chapter 6.

For many people, this part of the book is the most fun, because of the graphics and because it is the easiest to learn.

Part II Chapter 7–11

The emphasis here is on the "rest of Logo"—how Logo uses numbers, words, lists, inputs, outputs and conditionals. This part requires more concentration and study than Part I.

Chapter 7, 8 and 9 are the key chapters. You might read over these chapters lightly, then try out the ideas at the computer, then reread the chapters. At first, it is only necessary that you be able to understand the examples. Then, time and practice will help you start writing similar programs.

The section in Chapter 9 on "Recursion and OUTPUTs" is probably the most difficult part of the whole book. Once again, time and practice are the key ingredients towards understanding. It is often helpful to find someone else learning Logo and use the "box charts" to try to explain these new concepts to them.

Chapter 10 can be skipped if you're not interested in Logo's mathematics capability at this time.

Chapter 11 summarizes Logo's use of "conditionals" and includes some nice program examples.

Part III Chapter 12–15

The emphasis here is on programming projects.

Chapter 12 is the most important of the group. It provides an overview of ways to improve your program writing. This chapter should be gone over carefully and ideas tried out.

The remaining three chapters can be used according to your interests. These chapters offer examples of the types of programs that Logo can do.

If you're learning Logo in a noisy environment, either at home or in school, we suggest that you skim the sections or chapters that you'll be using before sitting down at the computer. Don't try to study the book at this time, because the best way to learn Logo is at the computer. However, it is helpful to get a rough idea of what to expect and what you might like to try out. Also, the book provides explanations that are sometimes hard to follow when the computer screen and friends beckon for your attention.

Probably the most important point to remember is this: If you're having fun, "playing" with Logo, then you're learning at maximum efficiency. However, if you begin to feel frustrated, then it's time to slow down, perhaps find someone to help you or reread the explanations in the book.

One of the nice side effects of studying the Logo language is that it demonstrates once again that learning is one of life's most enjoyable experiences.

CHAPTER 1

GETTING STARTED

CONCEPTS:

Apple setup for Logo
keyboard overview
loading Logo
introducing turtle graphics
Logo primitives
making changes to typed lines
writing procedures
creating procedures from procedures

PRIMITIVES:

SHOWTURTLE, FORWARD, BACK, RIGHT, LEFT, CLEARSCREEN, PENUP, PENDOWN, TO, END, EDIT, REPEAT

1.0 Introduction

Welcome to programming in Logo! In this first, introductory chapter, we'll explain how to get your Apple turned on, or as they say, "up and running" in the language and familiarize you with the keyboard controls in Logo. We'll develop your skills using the Logo commands to write and edit graphics programs.

To coin a new phrase, "a little bit of knowledge about Logo is a wonderful thing"; we think you'll be intrigued by Logo's capabilities after just this brief exploration. We hope you will enjoy yourself!

1.1 Hardware Overview: System Requirements for Logo

In order to run Logo, you'll need at minimum:

1. an Apple II, Apple II+ with 64k of memory, or an Apple IIe.
2. a single disk drive with DOS 3.3.
3. a television or video monitor, preferably with color, although a black and white, green, or amber screen is also good.
4. Apple Logo

A typical computer set-up is shown below. You might check to see that the power cord at the (left) rear of the Apple is plugged in; that the video monitor is connected to the (right) rear of the Apple; and that the disk controller card (usually in slot #6) linking the disk drive(s) to the computer is in place.

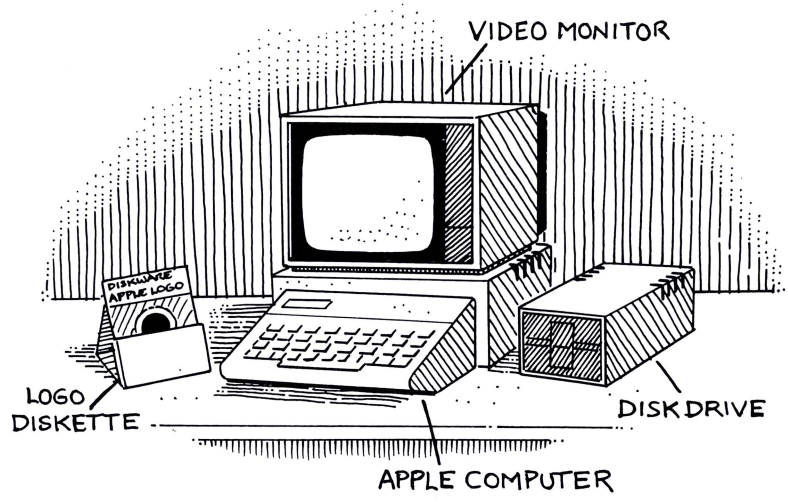


Fig. 1-1. Typical Apple computer setup for running logo.

1.2 The Keyboard

If this is your first experience with an Apple computer, you might want to spend a few minutes examining the keyboard. The keyboards for the

Apple II/Apple II+ and Apple IIe are shown below. You may be familiar with the placement of alphabet letters and numerals on the keyboard, as it is similar to typewriters. Computer keyboards do not allow you to substitute the letter "l" for the number "1," or letter "o" for zero. (Apple II and II+ users have a slash through the zero to make this distinction; Apple IIe users will find the number zero on the same key as the right parenthesis.) Be sure to use a number when you mean a number and a letter when you mean a letter. Apple, like all good computers, accepts no substitutions! The keyboard "touch" (go ahead!) is not unlike that of a typewriter, but without the attending "click."

In examining the keyboard, you will notice some differences from a typewriter. The primary difference is that there are additional keys not found on a typewriter. Take a minute to locate each of the following keys:

Apple II or Apple II+

ESC	(abbreviation for "escape")
CTRL	(abbreviation for "control")
REPT	(abbreviation for "repeat")
→	(right arrow)
←	(left arrow; also called "delete")
RESET	(the only key you DON'T want to hit!)
POWER	(when the light is on, the computer is on)
RETURN	(used as a carriage return, or as a signal to Logo to carry out your directions)

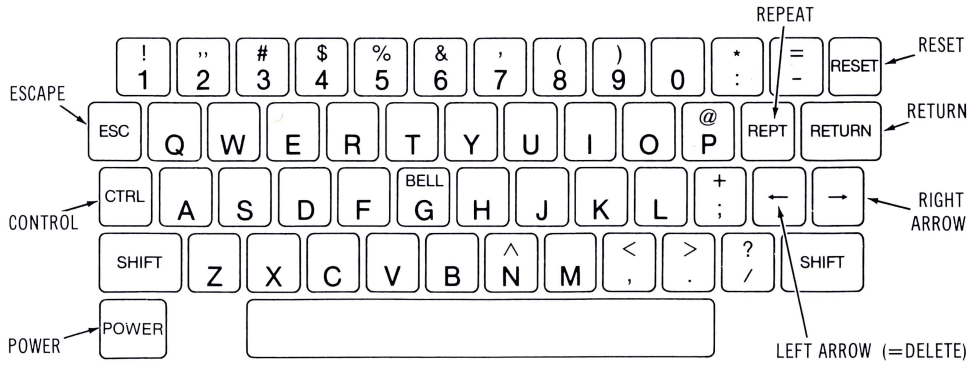


Fig. 1-2. Apple II, II+ keyboards.

Apple IIe

ESC	(abbreviation for "escape")
CONTROL	
→	(right arrow)
←	(left arrow)
[	(left bracket)
]	(right bracket)
RESET	(if you hold down the CONTROL key and press RESET, you'll exit the Logo language)
RETURN	(used as a carriage return, or a signal to Logo to carry out your directions)

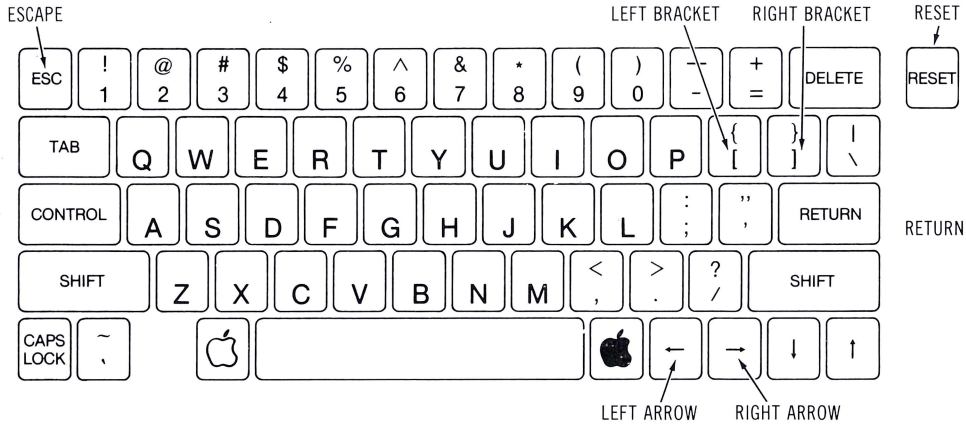


Fig. 1-3. Apple IIe keyboard.

The use of these keys varies with computer languages and programs. "Escape," for example, doesn't always get you out of a problem! In order to explain their use in Logo, and to have you begin experiencing this powerful language, let's get Logo up and running. For now it's enough to know where these keys are located.

1.3 Loading Logo

Take the Logo diskette out of its protective envelope, and hold it with your thumb on the label as shown in Fig. 1-4.

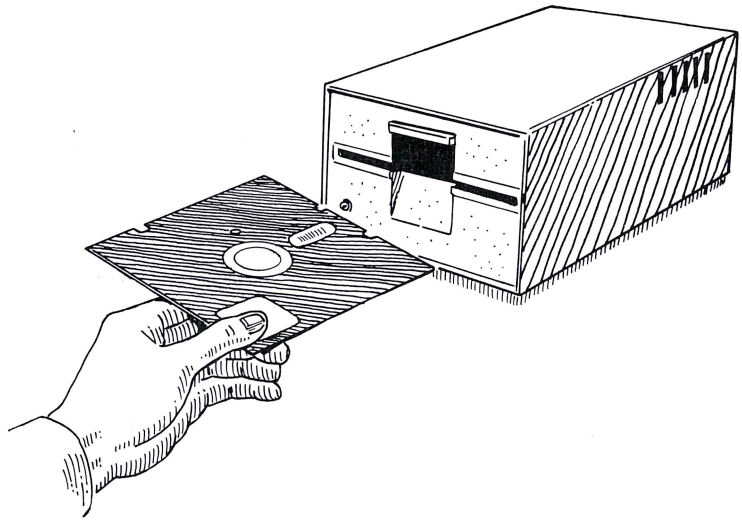


Fig. 1-4. Inserting Logo disk into the drive.

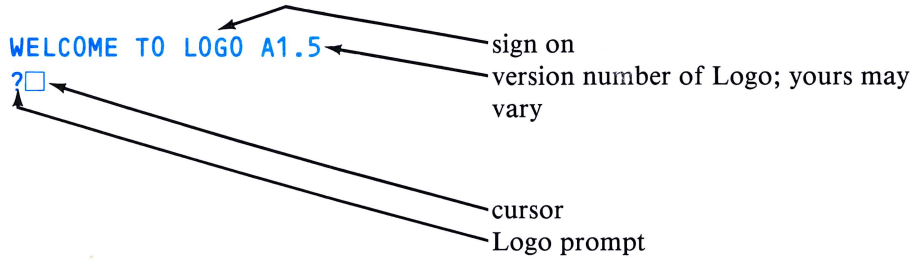
Open the disk drive door, and gently insert the disk until it is completely within the drive. Close the door of the drive. If you have two disk drives, insert the diskette into the drive marked DRIVE 1. Now turn on the monitor (the screen will be either blank or snowy at this point). With your left hand, turn the computer on by flicking the rocker switch at the rear left of the Apple. The power light on the lower left hand side of the keyboard should light up, as will the red light on the disk drive. You'll hear a soft whirring as the diskette turns in the drive, and "APPLE II" will be showing on the screen. You'll hear the disk drive whirr and stop, whirr and stop for a few seconds.

The red light on the drive will go off, and the screen will display the following:

```
PRESS THE RETURN KEY TO BEGIN


IF YOU HAVE YOUR OWN FILE DISKETTE,
INSERT IT NOW, THEN PRESS RETURN.
```

```
(C) LOGO COMPUTING SYSTEMS, INC. 1982
```

We will explain later how to create a file diskette. You do not need one at this point, so simply press the RETURN key. The screen will go blank and the drive will whirr as the information on how the language Logo works continues to be read from the disk into the computer's memory. You may think of it as a record being played on a stereo. When the information has been read in, you'll see the following on the screen:



Congratulations! The sign on "WELCOME TO LOGO" indicates that Logo is up and running (we say you've "booted" Logo). The question mark (?) is called the Logo prompt; it indicates that the system is awaiting your command. The flashing square after the prompt is called the cursor. It indicates where the next character would appear if you typed something on the keyboard.

At this point, our analogy of a record player needs to be modified. Unlike a stereo system, once the information is read into the computer, it stays there. All the information you need to run Logo is now in the Apple, and will stay there until you turn the machine off or (oops!) press RESET or CONTROL - RESET.

1.4 Using Logo Commands—An Introduction to Turtle Graphics

We'll begin our description of Logo by introducing you to Logo graphics, drawn by a creature on the screen known as the "turtle." Type in the

word "SHOWTURTLE" at the cursor point, and press the RETURN key. The screen will then look like this:

Turtle

?□

In the center you'll see a small triangle pointing up. This is the Logo turtle, which has had an interesting evolution. It began in the late 1960s as a small toy which moved in response to directions typed in on the computer keyboard. In the 1970s, computer graphics became available for Logo, and the turtle moved onto the screen.

The small triangle indicates that you're now in "turtle graphics," and can type in commands to draw lines and figures on the screen. The word "SHOWTURTLE" is called a "command"; it directs Logo to perform a task. In this case, it commands Logo to "Show the Turtle." The RETURN key is your prompt to Logo that you have finished typing your command.

Now type in the following command:

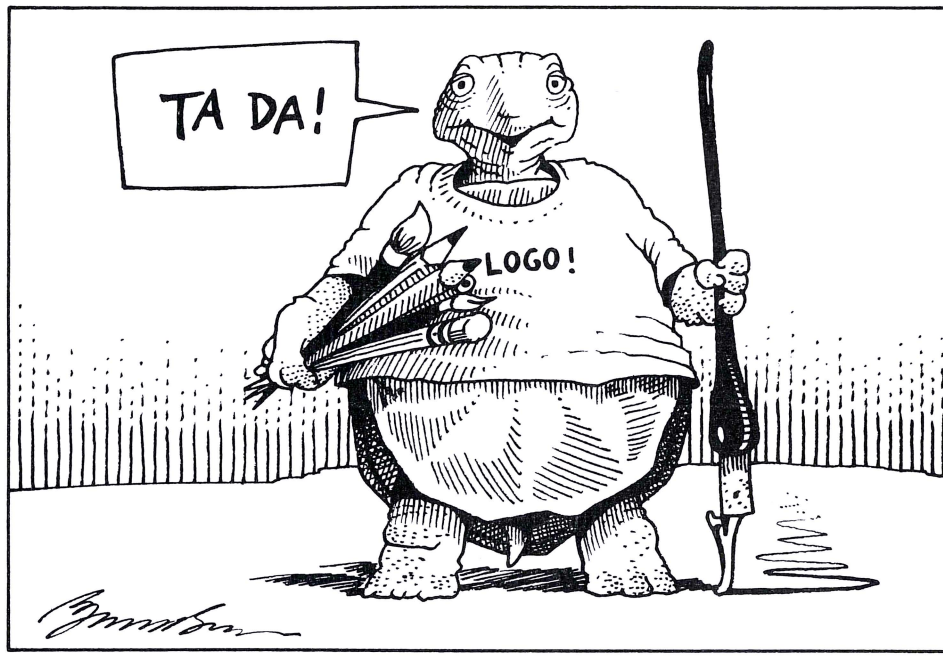
```
?FORWARD 50
```

and press the RETURN key. Be sure to leave a space between the command FORWARD and the number 50. On the screen, the turtle will have drawn a line forward 50 spaces!

In order to have the turtle move in the opposite direction, use the command BACK. For example,

```
?BACK 50
```

and RETURN will zip the turtle back to its original position. Note that its "tracks" are still displayed on the screen.



The turtle (and anyone programming it!) would get bored very quickly if all the turtle did was move up and down this line. Fortunately, that's not the case. The turtle will change its directions or heading with the commands

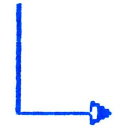
RIGHT and LEFT. For example, to change the turtle's direction 90 degrees to the right, type

```
?RIGHT 90
```

and press RETURN. If you now type

```
?FORWARD 30
```

and RETURN, the turtle will move 30 steps in that new direction:



In summary, there are four basic turtle moves: FORWARD and BACK move the turtle a specified number of steps; RIGHT and LEFT turn its heading a specified number of degrees. These commands, and many others in Logo, all have abbreviations, to make our typing simpler:

COMMAND	ABBREVIATION
SHOWTURTLE	ST
FORWARD	FD
BACK	BK
RIGHT	RT
LEFT	LT
CLEARSCREEN	CS

To clear the lines off the screen, type the command CLEARSCREEN (abbreviated CS) and press RETURN. When you do, you'll see the turtle back in its original position, and the familiar prompt and cursor at the lower left of the screen.

If you'd like to have the turtle move without making a line, use the command PENUP (abbreviated PU); when the turtle is in the place you want to make tracks, you can negate PENUP with PENDOWN (abbreviated PD). You can think of the turtle as riding on a pen, which can be raised (in PENUP) or lowered (PENDOWN).

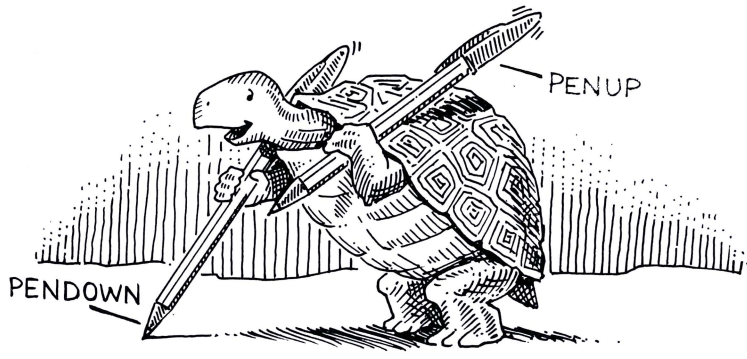


Fig. 1-5. PENUP vs PENDOWN turtle.

For example, try the following commands:

```
?PU
?RT 65
?FD 10
?PD
?FD 10
```

Experiment using the commands FD, BK, RT, LT, CS, PU, and PD until you're comfortable making turtle tracks on the screen. Don't worry about wearing the turtle out—this is a very aerobically conditioned turtle! (As in Fig. 1-6) And, unlike its other reptilian relatives, it has no problem moving its head 167 degrees to the left, followed by 89 degrees to the right, and so on.

1.5 On Line Editing

If you make a mistake in typing a command name, you'll get an "error" message. For example, type the following:

```
?FWARD 35
```

and press RETURN. Logo will respond by printing

```
I DON'T KNOW HOW TO FWARD
```

Similarly, if you type

```
?RIGHT22
```

Fig. 1-6. Aerobic turtle.

and don't leave a space between RIGHT and 22, you'll see

```
I DON'T KNOW HOW TO RIGHT22
```

printed on the screen. The reason you get these error messages is that Logo "knows" or understands some commands, like FD, RIGHT, etc., but if want it to FWARD something, you'll have to teach it what "FWARD" is. Logo understands RIGHT, but needs a space between the command word and the number following it. More on that later; let's first find out how to correct typographical errors on a line.

Mistakes on a line can be corrected if you note the error before pressing the RETURN key. (The RETURN key, as you've discovered, is your signal to Logo that you want the command carried out or executed. We won't be reminding you to push it after this.) For example, if you typed

```
?FORWARD 40
```

and you only wanted to go FORWARD 30 you can correct it as follows.

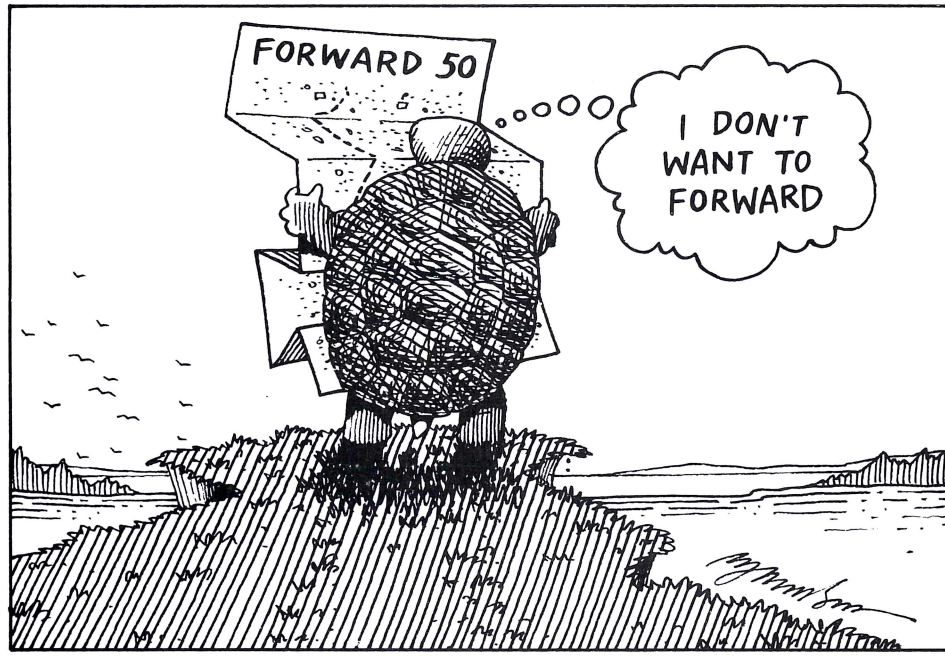
Hold down the key marked CTRL or CONTROL with your left hand and with your right hand depress and release the A key. This procedure, called "CTRL-A" (pronounced "control A"), moves the cursor to the beginning of the line. Now press the right arrow key. Note that it moves the cursor one space to the right. Continue striking the → key until you get it

to the number 4 (you may move it more quickly if you hold down the REPT key for Apple II and II+, or hold down the same key for the Apple IIe). When the cursor is over the 4, hold down the Control key, CTRL, and press the letter D smartly. This move, called "control-D," erases the number or letter directly under the cursor. Now type in the number 3. To move the cursor to the end of the line, strike the → key again. When you press RETURN, the corrected command will be executed.

There is another way to correct this type of typographical error. Using the same example,

```
?FORWARD 40
```

can be changed to FORWARD 30 by using the following process. With the cursor at the end of the line, strike the left arrow key twice. Notice that both the 0 and the 4 are gone; the ← is called the delete key. In your Logo kit you should find a small sticker with DEL for "delete" on it. Attach it on the ← key to remind you that this key erases anything under the cursor. When you've deleted the 0 and 4, type in 30, and execute the line. Apple IIe users might be tempted to use the DELETE key for this maneuver, but unfortunately it won't work!



There are a few other editing commands that you can use if you note your mistake before you've pressed the key marked RETURN. These special control sequences are listed below along with those we've just discussed.

Editing Keys

CTRL-A	moves the cursor to the beginning of the line
CTRL-D	deletes the character under the cursor
→	moves the cursor one space to the right
←	cursor moves left one position and erases character
CTRL-B	moves the cursor one space to the left or "back" but does not erase
CTRL-K	erases or "kills" all characters to the right of the cursor
CTRL-Y	restores the "yanked" characters erased by CTRL-K
CTRL-E	moves the cursor to the end of the line.

When we want you to do a control sequence, we'll use this format: CTRL-A. How you choose to use the combination of editing keys to correct errors is up to you. For example, to change

```
?FOXWARD 50
```

to

```
?FORWARD 50
```

any of the sequences below will work. Try each of them to confirm this for yourself.

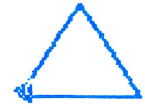
1. CTRL-A; → to X; CTRL-D; type letter R; → and REPT key to end of line.
2. CTRL-A; → to X; CTRL-D; type letter R; CTRL-E to end of line.
3. CTRL-B to W; ←; type R; CTRL-E to end of line.
4. ← to X; retype rest of line.
5. CTRL-B to W; CTRL-K; ← one space; type R; CTRL-Y.
6. CTRL-A; CTRL-K; and retype line.

Don't worry about memorizing these editing moves. Frequent use will make them feel familiar, and you can always refer to the summary of commands at the end of this chapter or in the appendix if you forget specific sequences. The tear-out Logo Startup Card should be kept in front of you. This card summarizes the major commands from the first four chapters.

1.6 Summary

Let's review what we've done so far. We've learned: to boot Logo; to enter the turtle graphics mode by ST; to change the turtle's directional heading by using RT or LT; to have it draw lines of specified length (by using FD, BK, PU, PD) in the direction chosen. We can wipe the screen clear by CS. We can also use the right and left arrow keys and the CTRL key in combination with others to edit commands.

To test your skills, try to direct the turtle to draw an equal-sided triangle on the screen, like the one below, with sides of 50 turtle steps.



How about trying a star? The one below is made up of two equal-sided triangles:

A hint or two . . . triangles with equal sides (isosceles) have outside angles of 120 degrees. Second hint . . . you'll need to use PENUP to prevent unwanted lines. Feel free to work on other designs and shapes before going on to the next section.

1.7 Creating Procedures

Now we're ready to begin programming in Logo. Logo programming consists of teaching Logo new commands based on terms or words it

already knows. For example, Logo understands commands such as RT, PK, BK, etc.—these commands are called "primitives." They are the basic building blocks of all programs. Primitives are combined to define new words, called "procedures."

Let's use some of the primitives we've explored so far to write procedures. We can think of procedures as adding new words to Logo's vocabulary. For example, to teach Logo to have the turtle draw a square, clear the screen (CS) and type in the following:

```
?TO SQUARE      You type this in and press RETURN
>□              Logo automatically responds with this symbol followed by the
                cursor
```

The word TO is a primitive, like FD, RT, and CS. Its effect is to direct Logo to accept a set of directions on how to do something, in this case how to make a SQUARE. The name of our procedure, SQUARE, follows the primitive TO on the first line of our program. Notice that Logo's response to your TO SQUARE direction was to display a > symbol. The change from ? to > on the prompt line indicates that Logo is expecting (and accepting) a set of instructions.

Now let's type in the directions on how to make a square. To make a square with side of 50, we need to go forward 50, turn right 90 degrees, forward 50, right 90, FD 50, RT 90, and so on. Type in each line and press RETURN:

```
?TO SQUARE      Procedure name
>FD 50          Moves the turtle forward 50 steps
>RT 90          Turns the turtle 90 degrees to the right
>FD 50          Moves the turtle forward 50 steps
>RT 90          Turns the turtle 90 degrees to the right
>FD 50          Moves the turtle forward 50 steps
>RT 90          Turns the turtle 90 degrees to the right
>FD 50          Moves the turtle forward 50 steps
>RT 90          Turns the turtle 90 degrees to the right
>END            Ends the procedure definition
```

In the last line of the program, we type in the primitive END. This indicates to Logo that we've finished giving directions. Logo responds by printing

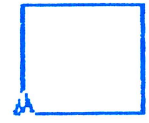
```
SQUARE DEFINED
?□
```

Notice that the > was replaced with our familiar ? and flashing cursor, signaling that Logo is finished learning how to SQUARE and awaits our next direction.

If you now type the command

```
?SQUARE
```

and press RETURN, Logo will draw the following:



Aha! Logo now knows how to SQUARE, just as it understands CS, ST, and the other primitives in its vocabulary. As we'll soon see, SQUARE can now be used in writing other procedures, just as if it were a primitive.

For now, though, let's look at our program SQUARE more closely. Notice that SQUARE consists of four sets of FD 50 RT 90. Logo allows you to put more than one command on a line. Thus, an alternative to our first program, SQUARE, would be as follows:

```
?TO SQUARE.1          Procedure name
>FD 50 RT 90         Moves the turtle forward 50 steps, and turns it right 90 degrees
>FD 50 RT 90
>FD 50 RT 90
>FD 50 RT 90
>END
```

Run the program SQUARE.1 by typing in SQUARE.1 and press RETURN. Note that it gives the same drawing as SQUARE. Whether you write single or multiple commands on a line is largely a matter of personal style, sort of like whether you prefer your cottage cheese with or without ketchup.

You may be curious about the name we gave to our second procedure, "SQUARE.1." In general, it's useful to give procedures names that are meaningful to us; Logo wouldn't care if we called this procedure ELEPHANT or DIMELKJ—but we're interested in making the names communicate the essence of the procedure to us. Apple Logo does, however, have some restrictions about names. It won't let you use primitives or already defined procedures as names. For that reason, we couldn't call our second

procedure for drawing a SQUARE by the same name. Logo has two other restrictions about names. It requires them to be 30 characters or less, and it requires their names to be one continuous string or flow of characters. Thus Logo sees SQUARE.1 as one word; we use a period between the E and the number 1 to make it easier for humans to read the title.

Let's examine the SQUARE.1 program more closely. Notice that it consists of four sets of FD 50 and RT 90. Is there a way to simplify all these directions? Yes, Logo does have a way to indicate that an action is to be replicated, using the primitive command REPEAT. We'll group the portion to be repeated inside brackets which look like this: [ ]. Note that these are *not* the same as parentheses: ( ). Brackets are produced by holding down the SHIFT key and pressing letter N for the left bracket and M for the right on Apple II or II+ machines; Apple IIe has bracket keys. Decals for the bracket symbols for Apple II and II+ are found in your Logo kit for affixing to the N and M keys.

Now type in the following program:

```
?TO EASYSQUARE              Our procedure name
>REPEAT 4 [FD 50 RT 90]  Be sure to include spaces as shown
>END
```

Clear the screen with CS and type EASYSQUARE to run the program. Note that it gives you the same figure as SQUARE and SQUARE.1, with a lot less typing!

Suppose we want to modify EASYSQUARE by making the drawing larger. We could write another program and use a different value for FORWARD, but there is a way to edit the procedure we've written. We can change the procedure for EASYSQUARE by going into the Logo editor. Type

```
?EDIT "EASYSQUARE
```

and press RETURN. Note that you enter the editor by typing EDIT followed by a space, then a quotation mark (“), and the name of the procedure you wish to edit. You don't need a quote (”) at the end of the procedure line (after the “E”).

The screen changes, and the lines seen at the top of the next page are displayed.

The line at the bottom reminds you that you're in the Logo editor. The cursor will be flashing over the first letter, “T.” You can alter your program using your choice of the right and left arrow keys, CTRL-A, -B, -D,

```
TO EASYSQUARE
REPEAT 4 [FD 50 RT 90]
END
```

LOGO EDITOR

-E, -K, or -Y commands described earlier. In addition, there are other commands available to you in the edit mode: CTRL-N moves the cursor down to the next line, and CTRL-P moves the cursor up to the previous one. Hitting the RETURN key at the end of a line will "open up" a new (blank) line. CTRL-O will also open up a new line. (It's best to use CTRL-O if the cursor is at the beginning of a line.) These open lines can be closed up by pressing the left arrow (delete) key. When you've finished editing, you exit the editor with the sequence CTRL-C. The control sequences for use in the editor are listed below. Note that our list contains both "new" and "familiar" keys.

Control Keys for Use in the Editor

CTRL-N	moves the cursor to the next line down
CTRL-P	moves the cursor up to the previous line
CTRL-O	opens up a new line at the cursor location
CTRL-C	exits the editor saving all changes
CTRL-A	moves the cursor to the beginning of the line
CTRL-D	deletes the character under the cursor
→	moves the cursor one space to the right
←	cursor moves left one position and erases character
CTRL-B	moves the cursor one space to the left or "back" but does not erase

CTRL-K erases or "kills" all characters to the right of the cursor
CTRL-Y restores the "yanked" characters erased by CTRL-K
CTRL-E moves the cursor to the end of the line.

Use your choice of editing commands to change the program to read as follows:

```
TO EASYSQUARE
REPEAT 4 [FD 80 RT 90]
END
```

This was 50; change to 80

LOGO EDITOR

When you've finished, exit the editor by the command

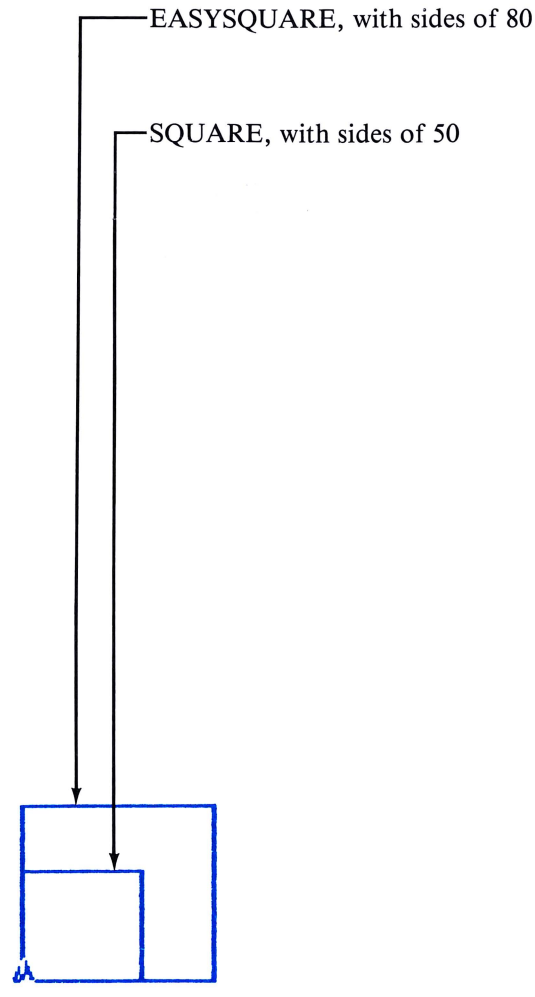
CTRL-C

EASYSQUARE DEFINED will be printed at the top of the screen, followed by the ? and cursor. Now type

```
?EASYSQUARE
```

and press RETURN

Confirm that EASYSQUARE is indeed larger by now typing SQUARE, and press RETURN. The screen will look like this:



What we've done thus far is to "teach" Logo three new procedures: SQUARE and SQUARE.1 and EASYSQUARE. These new procedures are now in Logo's temporary memory inside the Apple, and can be used just as primitives in defining new words for as long as the Apple is turned on. We'll see how this is done in the next section.

1.8 Creating Procedures from Procedures

Once we have taught Logo procedures based on the primitives it knows, we may treat these procedures as though they were primitives. That is, we can write new procedures which include, or "call," those that we've taught Logo. This allows us to easily write more complex procedures, as we'll describe in the next section.

Let's say we want a diamond on the screen instead of a square. A diamond shape is just a square, rotated 45 degrees on its axis.

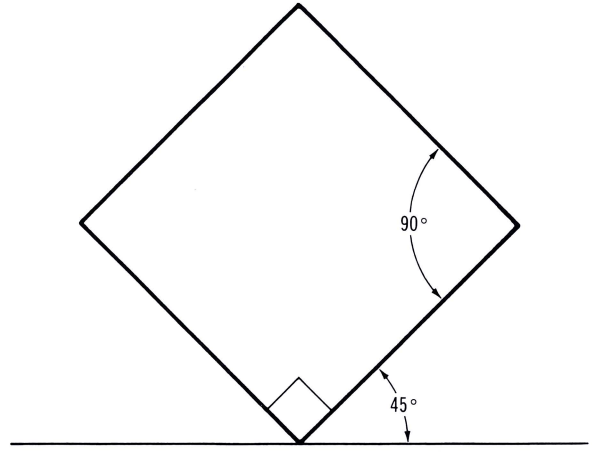


Fig. 1-7. Diamond geometry.

To draw the figure, the following program would work:

```
?TO DIAMOND      Procedure name
>LT 45           Rotates the turtle's head 45 degrees left
>FD 80 RT 90
>FD 80 RT 90     Draws a square
>FD 80 RT 90
>FD 80 RT 90
>RT 45           Rotates the turtle's head back to the
>END             original position
```

But before you type in DIAMOND, let's think about another way to write the procedure. We could use the REPEAT primitive to direct Logo to draw a square:

```
?TO DIAMOND
>LT 45
```

```
>REPEAT 4 [FD 80 RT 90] Draws a square
>RT 45
>END
```

But there is an even more elegant way. Note that REPEAT 4 [FD 80 RT 90] is the definition for EASYSQUARE. In Logo, you may use already defined procedures in defining new procedures.

Thus, the following program will also work:

```
?TO EASYDIAMOND
>LT 45
>EASYSQUARE
>RT 45
>END
```

Try it!

Let's modify EASYDIAMOND by making it smaller. How will we change the length of the sides that make up the diamond? You're right if you said we'll have to edit EASYSQUARE. EASYSQUARE is the procedure that determined the size of our drawing. Enter the editor again with

```
?EDIT "EASYSQUARE
```

and press RETURN. Change the length of the sides from 80 to 55. Then press CTRL-C to exit the editor. Now type

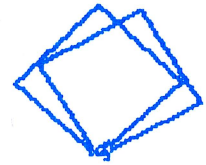
```
?EASYDIAMOND
```

and press RETURN. Note that it is indeed a different size than it was before.

Let's try a further modification. Let's repeat the EASYDIAMOND drawing while rotating it through a number of degrees. To draw our diamond, rotate it 15 degrees to the right, and then draw it again, you can use the following procedure:

```
>TO ROTATE
>EASYDIAMOND This draws our diamond shape
>RT 15       This rotates the turtle's direction
>EASYDIAMOND This draws our diamond shape again
>END
```

Now run the program ROTATE. The figure you see will look like this:



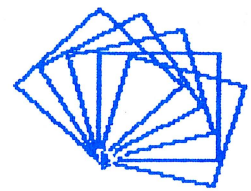
Let's say we wanted to repeat the pattern more times. (Diamonds, after all, are a programmer's best friend . . .) To rotate the diamond five times, we could edit ROTATE to the following:

```
?TO ROTATE
>EASY DIAMOND
> RT 15
>EASYDIAMOND
>RT 15
>EASYDIAMOND
>RT 15
>EASYDIAMOND
>RT 15
>EASYDIAMOND
>RT 15
>EASYDIAMOND
>END
```

You may already have figured out an easier way. We could use REPEAT to indicate that we want Logo to draw the shape, rotate 15 degrees, etc. Let's use REPEAT in a program called EASYROTATE:

```
>TO EASYROTATE
>REPEAT 5 [EASYDIAMOND RT 15]   Draws a diamond, rotates 15 degrees
>END                            Repeats this five times
```

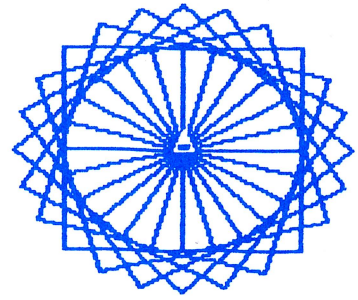
Run EASYROTATE by typing it in and pressing RETURN. The drawing you'll see is as follows:



Let's finish our drawing off by rotating the diamond shape through a full 360 degrees. How many repeats will this take? Let's see . . . 360 divided by 15 is 24. So, our EASYROTATE program needs to be edited to read as follows:

```
>TO EASYROTATE
>REPEAT 24 [EASYDIAMOND RT 15] Change 5 to 24
>END
```

When you now run EASYROTATE, you'll see the following displayed on your screen:



1.9 Summary

Congratulations! We've learned a lot about Logo and Apple II operations in this first chapter (particularly if you've done it all in one sitting . . .) A summary of commands we've learned so far, and some review questions to test your understanding follow. When you're finished, be sure to shut down the system completely. Remove the Logo diskette from the disk drive and replace it in the protective envelope. Then, turn off the video monitor. (Most of us do this before turning off the Apple II; we're less likely to forget and leave it on!) Then, turn off the Apple computer by depressing the rocker switch on the back left corner. We hope you enjoyed this introduction, and we look forward to working with you as you learn more about this fascinating language.

1.10 Summary of Commands

1. Basic Turtle Motion Commands (with typical value 50)

FORWARD	FD 50
BACK	BK 50
RIGHT	RT 50
LEFT	LT 50

2. Controlling the Turtle

PENUP	PU
PENDOWN	PD
SHOWTURTLE	ST
CLEARSCREEN	CS

3. Cursor Motion Keys

CTRL-A	Move to beginning of line
CTRL-B	Move back one space
CTRL-D	Delete character under cursor
CTRL-E	Move to end of line
CTRL-K	Delete to end of line
CTRL-Y	Replace last deletion using CTRL-K
→	Move right one space
←	Delete back one space

4. Creating a Procedure

TO (name)	Puts you in procedure mode
END	Leaves procedure mode

5. Editing a Procedure

EDIT "(name)	Puts you in editor
CTRL-C	Leaves editor
CTRL-O	Opens new line above cursor

CTRL-P	Moves cursor up a line
CTRL-N	Moves cursor down a line

Note: To open a new line above the cursor, it is best to place the cursor at the beginning of the line with CTRL-A, then hit CTRL-O.

1.11 Review Questions and Exercises

Review Questions

1. What does the Logo "prompt" look like? What does it mean?
2. What is the purpose of the "cursor"?
3. Why won't the following Logo commands work?
   a. RIGHT90
   b. FWD 50
   c. REPEAT [FD 10 RT 39]
4. What is a procedure?
5. Can you give a procedure the name "BACK"? How about the name "BOX"?
6. What happens if you type TO BOXER and the procedure BOXER is already defined? Will the editor erase the original BOXER procedure and start over?
7. When you are "in the editor," what key sequence is used to move the cursor . . .
   a. back one character without erasing the character?
   b. down one line?
   c. to the beginning of the line?
   d. to the end of the line?

Exercises at the Terminal

1. a. Create a procedure called LONGBOX that draws a short, long box shape.
   b. Place LONGBOX in the procedure EASYROTATE (in this chapter) in place of EASYDIAMOND.
   c. Try another shape.
2. Can you draw a circle?
3. a. Can you draw 4 separate boxes on the screen so that they don't touch each other?

b. Can you use REPEAT to do the same thing? (This gets a little more involved, since it requires several commands in the REPEAT brackets.)

4. If you really want to get tricky, try running two REPEAT commands on the same line, one inside the other, something like this:

   REPEAT 4 [REPEAT 3 [BOX RT 10] RT 120 FD 30]

   Explore the REPEAT command with more variations.

5. Practice editing procedures by changing the values of inputs, such as changing FD 60 to FD 40. Also, try adding more lines (commands) to existing procedures (like EASYROTATE).

Answers to Review Questions

(1) The Logo prompt is a question mark ("?"). The prompt indicates that Logo is ready, and is expecting you to type something. (2) The cursor tells you where the next character typed will be placed on the screen. (3) a. RIGHT90 needs a space between RIGHT and 90, like this: RIGHT 90. b. FWD 50 needs to be corrected to FD 50 or FORWARD 50. c. REPEAT [FD 10 RT 39] needs a number input after REPEAT, like REPEAT 4 [FD 10 RT 39] or REPEAT 25 [FD 10 RT 39]. (4) A procedure is a set of directions or commands which has been typed into Apple's memory. When we have finished typing in our procedure, Logo responds to the procedure name as though it were a primitive. In essence, when we write procedures, we "teach" Logo new words. (5) Logo would not allow you to call a procedure BACK as that is the name of a primitive. BOX would be okay, though. (6) If the procedure BOXER is already defined and in Apple's memory, you'll get an error message if you try to define it again ("BOXER IS ALREADY DEFINED"). The editor will not automatically erase the original procedure. (7) a. CTRL-B b. CTRL-N c. CTRL-A d. CTRL-E

CHAPTER 2

"TURTLEVERSE": THE TURTLE'S UNIVERSE

CONCEPTS:

command levels in Logo
the three screen modes in Logo
turtle boundaries
adding color
writing text
Logo as a calculator

PRIMITIVES:

*FULLSCREEN, TEXTSCREEN, SPLITSCREEN, WRAP, WINDOW, FENCE, HOME, POS, SETX, SETY, XCOR, YCOR, PRINT, SETPC, SETBG, PENCOLOR, BACKGROUND, +, −, *, /*

2.0 Introduction

In Chapter 1, we learned how to "boot" Logo into the Apple computer, how to enter turtle graphics and move the turtle, and how to teach Logo new words built up from its vocabulary of primitives.

If the theme of our first chapter was "introducing the turtle," Chapter 2 has as its central idea "turtle management." We'll explore how to move between different screen modes in Logo and how to determine the size of

the turtle's universe under different conditions. We'll also learn how to add color to our video ecosphere. In addition, we'll see how Logo can print text and function as a calculator.

Before we take a long look at our reptilian playing field, let's get an overview of exactly what Logo is doing at each moment in time. We'll do this by introducing the concept "Command Level."

2.1 Command Level

When we boot Logo and get the welcome sign on the screen,

```
WELCOME TO LOGO
?□
```

we are at the Command Level. Logo is waiting for our next command. The Command Level is always indicated by the question mark prompt (?). The Command Level can be thought of as the "control center," the pivotal point about which Logo revolves. There are three ways to temporarily leave the Command Level and we have already experienced each of them. From the Command Level, we can enter the "To Level," the "Edit Level," or the "Procedure Level" as shown in the figure below.

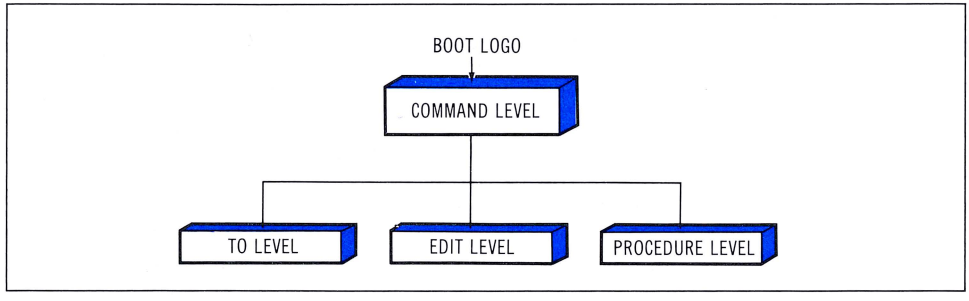


Fig. 2-1. Levels of command in Logo.

2.1.1 The TO Level

In Chapter 1, we learned how to group primitives together to write procedures in Logo. We found that procedures begin when we type in the command, TO. When we've finished writing our set of instructions, we indicate we're done with the primitive END. That's Logo's cue to take us back to the Command Level.

2.1.2 The EDIT Level

We also saw how to edit procedures that we've written. To enter the Edit Level, we type EDIT followed by quote marks and the procedure name. When we've finished our editing, we use the command, CTRL-C, to go back to the Command Level.

2.1.3 The Procedure Level

The third level, the Procedure Level, begins when we direct Logo to execute a procedure we've typed in. We enter that level by typing in the name of the procedure, and press the RETURN key. Logo executes the procedure and returns us to the Command Level when it encounters the END statement. (We'll see later in this book that there are other ways of stopping a procedure and returning to the Command Level.) Let's add to the diagram the ways we move between the three levels.

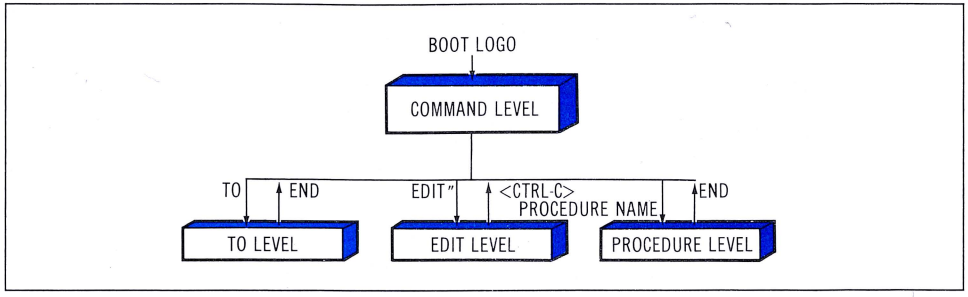


Fig. 2-2. Entering and exiting levels in Logo.

We'll now explore the three screen modes available from the Command Level. In addition, we'll continue practicing moving into, and exiting from, the To, Edit, and Procedure Levels. You'll find, with practice, that it is fun to take command of Logo.

2.2 Screen Modes In Logo

There is a terrible old joke that goes, "When is a door not a door?" The answer is, "When it's ajar . . . " We can consider the video display screen an analogous situation. Sometimes it's a turtle yard with an area where we can display what we type, sometimes the screen is given over exclusively to the turtle, and sometimes the screen becomes a blackboard for us to write text and perform calculations. When we boot Logo into Apple, we have a choice of three video display modes, as shown below.

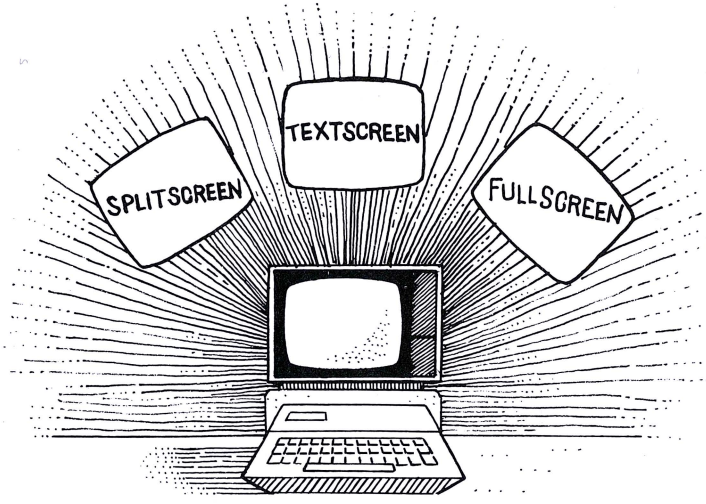


Fig. 2-3. Screen modes in Logo.

We'll begin by working with the SPLITSCREEN and FULLSCREEN modes; TEXTSCREEN will be covered later in this chapter.

2.2.1 SPLITSCREEN

In Chapter 1, we found that we could enter the world of turtle graphics with the command SHOWTURTLE (abbreviated ST). If we call the turtle with the command ST and press RETURN, the screen changes to look like this:

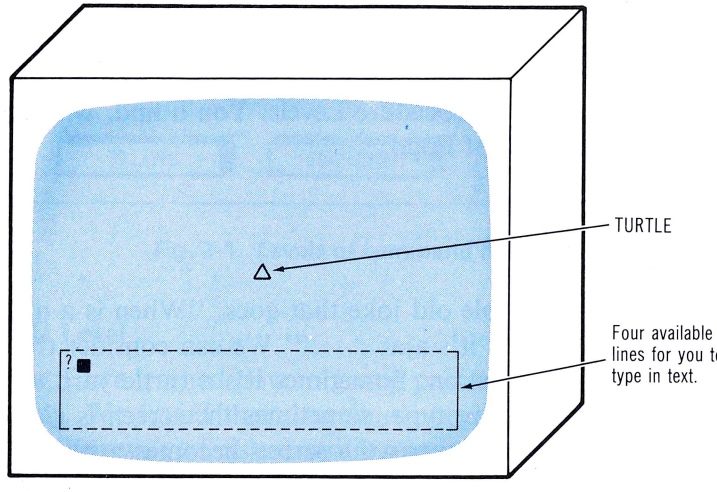


Fig. 2-4. Video display screen showing turtle.

When we enter turtle graphics using ST, we enter the screen mode called SPLITSCREEN; the screen is literally split into a top turtle field and a lower area of four lines which will display what is typed at the keyboard. The cursor and prompt are on the first line of this available space. SPLITSCREEN is a very convenient screen mode; we can watch the turtle move, and also see what we're typing. It does, however, place some limitations on the size of our drawings, as we'll see.

We entered SPLITSCREEN with ST. There are several other ways to enter this mode, as listed below.

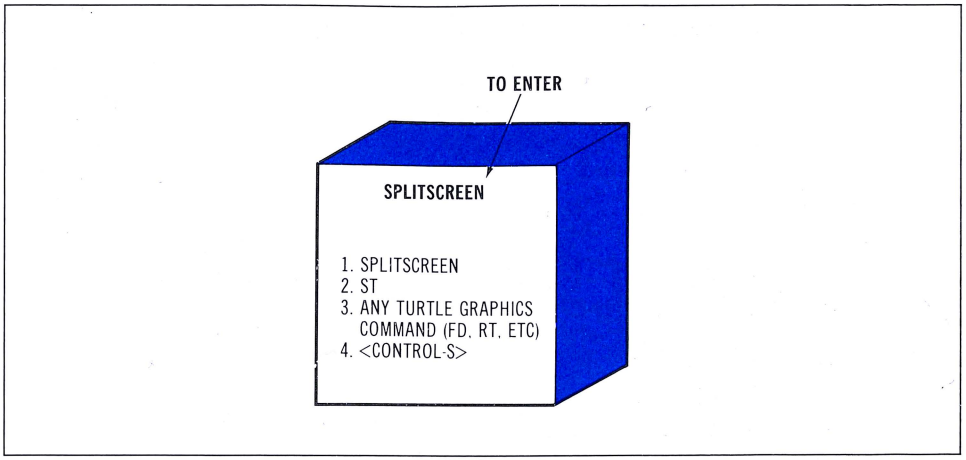


Fig. 2-5. Four ways to enter SPLITSCREEN.

Any turtle graphics command, such as FORWARD 50, or RIGHT 90, will place Logo in the SPLITSCREEN mode. The CONTROL-S sequence is different from turtle graphics commands in that is a "pure keyboard" command, i.e. it can't be used in a procedure. You used a CONTROL-C sequence to exit the editor; CONTROL-S works the same way. Depress the key marked CTRL (Apple II or II+) or CONTROL (Apple IIe) and hold it down. Now hit the key marked S. You did it—you're now in SPLITSCREEN! (or still there if you used ST)

In Chapter 1, we found that commands like FD, RT, PU, PD, and BK could be used alone, or grouped together to form procedures. Let's type in the procedure we wrote to draw a diamond, and use that shape to explore the turtle's world in SPLITSCREEN. (In Chapter 4, we'll show you how to save procedures to your disk. Then we'll be able to load them into the computer memory every time we want to use them and not have to type them over again.)

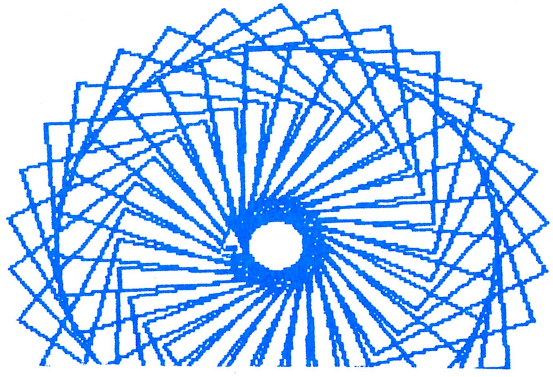
To instruct Logo how "to diamond," type in the following:

```
?TO DIAMOND                    procedure name
>LT 45                         turns left 45 degrees
>REPEAT 4 [FD 80 RT 90]        draws a square with sides of 80
>RT 45                         turns right 45 degrees
>END
```

Now we can call DIAMOND by typing its name at the Command Level and pressing the RETURN key. The figure should look like the one we drew in Chapter 1. Now let's write a procedure to have DIAMOND rotate through 360 degrees, using the REPEAT command:

```
?TO DIAMOND.PATTERN
>REPEAT 30 [DIAMOND FD 4 RT 12]  draws our DIAMOND, goes
>END                             FORWARD 4 steps, and turns
                                 RIGHT 12 degrees
```

After typing in this procedure, call DIAMOND.PATTERN. When the drawing is finished, the screen should look like this:



Oops! The four lines of text have hidden the bottom part of the diamond drawing. We need to "open" up that part of the video display to the graphics field. To do this, we need to leave the SPLITSCREEN mode and go into FULLSCREEN.

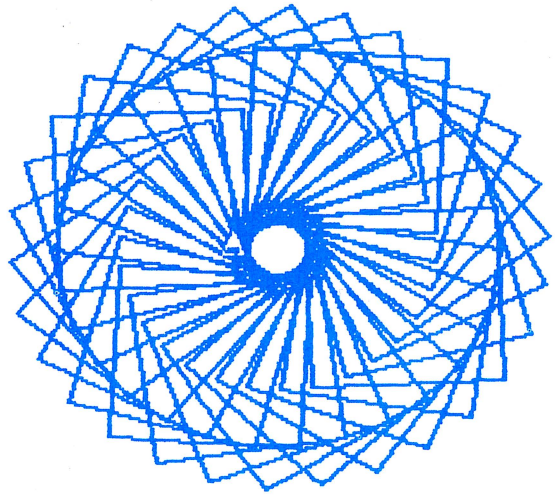
2.2.2 FULLSCREEN

The second display mode, FULLSCREEN, devotes the entire video screen to turtle graphics. To enter that mode, use the command

FULLSCREEN. To try this with the "cut off" drawing still on the screen, type in

```
?FULLSCREEN
```

and press RETURN. Voila! We can now see the bottom of the drawing. Notice that the cursor and prompt have disappeared from view; the entire, full screen is now devoted to the turtle.



There are two ways to enter this full screen video mode, as listed below:

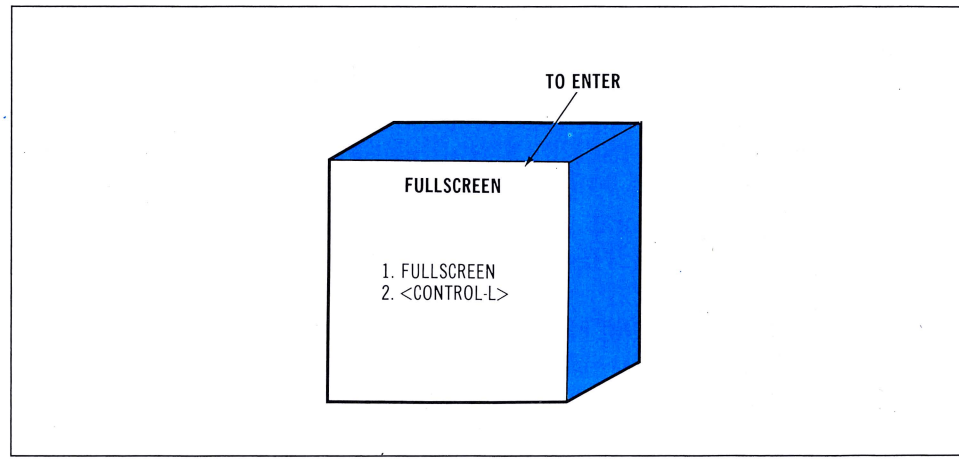


Fig. 2-6. Entering FULLSCREEN.

FULLSCREEN is both a keyboard and procedural command; CTRL-L, like CTRL-S, is a keyboard command only.

If we type in a command while we're in FULLSCREEN, we won't be able to see the command itself, but it will be "seen" by Logo and executed. For example, type in the command CS and press RETURN. The screen will clear. In order to see the diamond drawing again, type DIAMOND.PATTERN and press RETURN.

The full screen mode is admittedly quite nice for the visual antics of the turtle, but a little unnerving for human programmers who'd like to see what they're typing. Most of us move back and forth between FULLSCREEN, SPLITSCREEN, and as we'll see later, TEXTSCREEN, depending on what we're doing. To exit the FULLSCREEN mode, and go back to SPLITSCREEN, use any of the following:

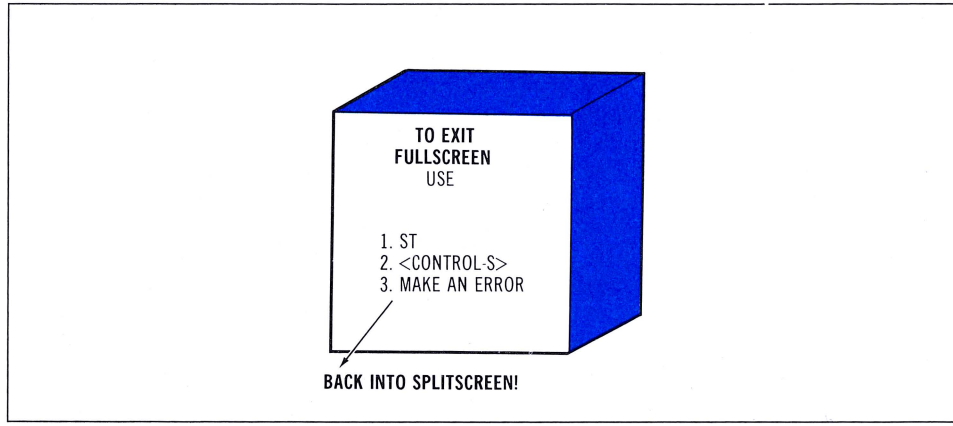


Fig. 2-7. Exiting FULLSCREEN.

We've used the first two directions, ST and CTRL-S. The other way of reentering SPLITSCREEN is to make an error or mistake (which isn't too difficult since we can't see what we're typing!) If, for example, we type in FOWWRD 50 and press RETURN while in FULLSCREEN, the video automatically goes back to SPLITSCREEN, with the error message

```
?I DON'T KNOW HOW TO FOWWRD
?
```

on the top SPLITSCREEN line. Our familiar cursor and prompt are again back in view.

We've seen that two of the video modes, SPLITSCREEN and

FULLSCREEN, are primarily worlds of the turtle. What is the size and shape of the turtle's universe? How far FORWARD and BACK can the turtle move? What happens when the turtle gets to the edge of its universe? Can its domain be expanded? Contracted? Like many cosmological questions, the answers are best obtained by experimentation and observation, as we'll see in the next section.

2.3 Turtle Boundaries

In section 2.2, we looked at two modes of video display for turtle graphics, SPLITSCREEN and FULLSCREEN. Within these two modes, there are three primitives which control the size and shape of the turtle's world. The primitives are WRAP, WINDOW, and FENCE. Let's add these primitives to our turtle management structure drawn earlier:

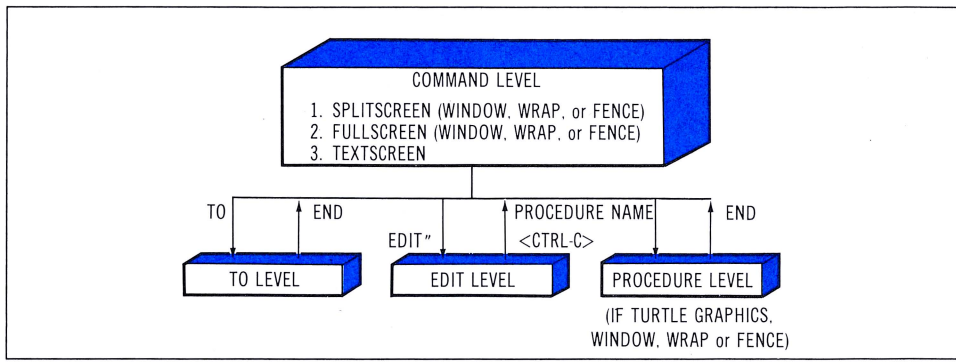


Fig. 2-8. Expanded command level showing screen modes and boundaries.

In the next three sections, we'll see how to enter and exit the three geographical areas, WINDOW, WRAP, and FENCE, and determine their relative size and structure.

2.3.1 WRAP

Let's begin by taking the DIAMOND.PATTERN procedure written earlier and modifying it. Instead of having the turtle draw a diamond and rotate it, we'll have it draw the shape and then move FORWARD and draw it again. Edit DIAMOND.PATTERN so that it looks like this:

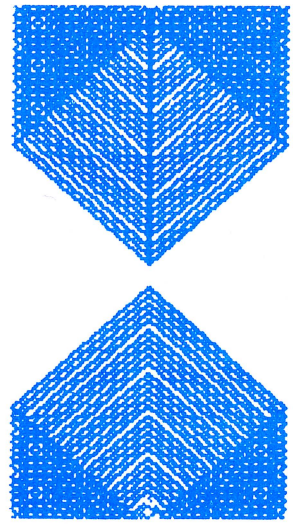
```
TO DIAMOND.PATTERN
```

```
>REPEAT 30 [DIAMOND FD 44]  we took out the RT 12 turn
>END
```

Now call the edited DIAMOND.PATTERN by typing

```
?DIAMOND.PATTERN
```

and press RETURN. Instead of rotating, the drawing now moves forward (up) toward the top of the video screen. Then it . . . what's this? We now see it coming up from the bottom! When it has finished the 30 repetitions, the screen will look like this:



What happened is that the turtle has "wrapped" itself around the video screen, in much the same way as a leashed dog might go around a tree. Does the turtle also wrap in the other direction? To find out, modify the diamond drawing to go horizontally.

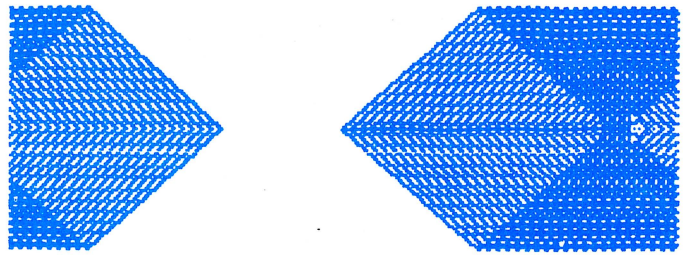
```
TO DIAMOND.PATTERN.HORIZONTAL
>RT 90                          This will turn the drawing on its side
>REPEAT 30 [DIAMOND FD 4]
>END
```

If the name of the procedure seems too long to type, you can either make it

shorter or create an abbreviation. For example, here is an abbreviated version:

```
?CS
?TO DPH
>DIAMOND.PATTERN.HORIZONTAL
>END
```

Now call DIAMOND.PATTERN.HORIZONTAL or DPH. The screen will look like this:



The turtle does, indeed, wrap around the screen in both directions. We call (appropriately enough) this state of the turtle's universe WRAP. When we enter turtle graphics, we are automatically in WRAP.

Let's see if we can determine the size of the WRAP universe by using Logo primitives that print the exact location of the turtle on the screen. These primitives are as follows:

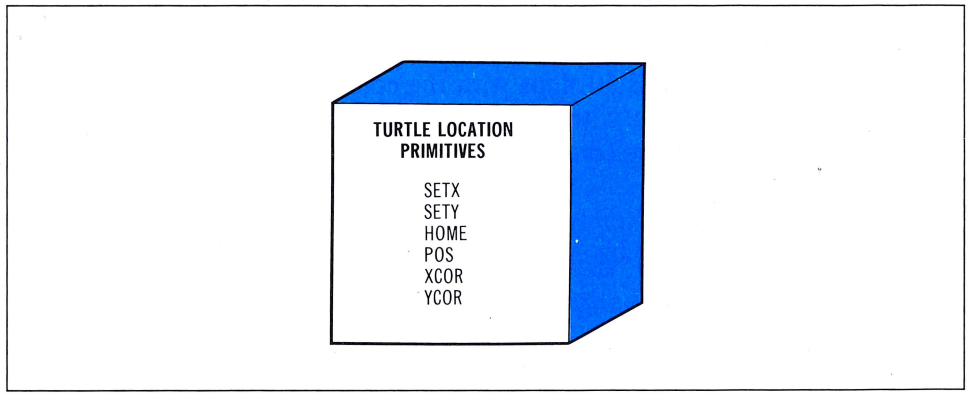


Fig. 2-9. Turtle location primitives.

In order to visualize the effect of these primitives, imagine a "Cartesian coordinate" system superimposed on the video screen.

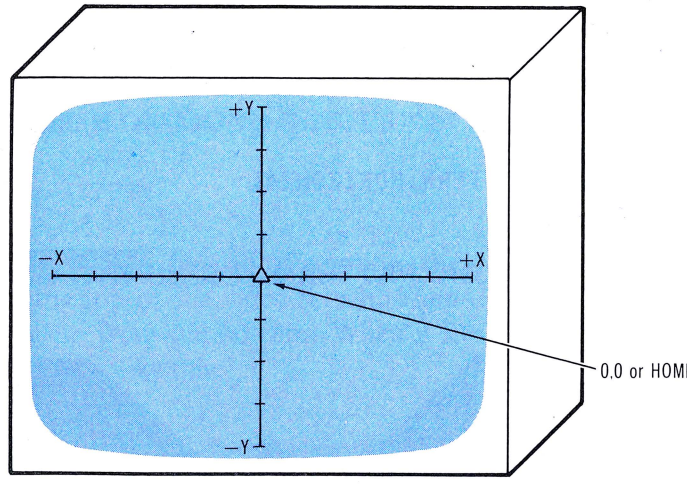


Fig. 2-10. Cartesian coordinate system on video screen.

The center of the Cartesian coordinate system is the point marked "0,0." That point is HOME to the turtle. When the turtle is first called with a graphics command, it appears at the HOME position. The primitive HOME can also be used to send the turtle back to (0,0) position at any time.

Horizontal movement on the Cartesian coordinate system is called the "X" direction; to the right of the 0,0 point is positive or +X; to the left of 0,0 is −X. Vertical movement is along the "Y" axis; above the 0,0 point is +Y; below it is −Y.

Let's try an experiment. Clear the screen using CS and move the turtle along the Y axis 50 steps with the command FORWARD 50. To display the turtle's location with respect to the X and Y axis, ask Logo to PRINT its position with the command

```
?PRINT POS          You type this in
0., 50.             Logo responds with this
?□
```

The response "0., 50" indicates that the turtle has not moved with respect to the X axis, but is 50 steps forward of the Y = 0 spot. POS and PRINT are both primitives. POS is a special kind of primitive called an "operation," which means it outputs a value that must be used with a primitive like PRINT. We'll explain operations in Chapter 3. For now, let's try a little more moving on the screen. Move the turtle

```
?BACK 100 RT 90 FD 60
```

The turtle will have moved to a spot in the lower right hand corner of the screen. Again, we can ask Logo to display the turtle's exact location with PRINT and POS:

```
?PRINT POS
60., -50.
?□
```

Two other primitives useful in this phase of turtle management are the commands SETX and SETY. These two primitives direct the turtle to a specific site on the coordinate system. Clear the screen (note that the turtle moves back to HOME, or the 0,0 position) and direct Logo to place the turtle 50 spaces to the left with the command:

```
?SETX -50
```

The turtle moves 50 steps on the X axis; its position with respect to the Y axis has not changed. To move the turtle on the Y axis, use the primitive SETY with a number input, as shown below:

```
?SETY 35
```

Now direct Logo to PRINT POS. It will respond by displaying the current coordinates:

```
?PRINT POS
-50., 35.
?□
```

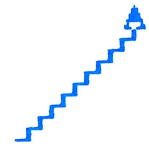
Two other primitives allow us to monitor and direct the turtle's movement on the screen. We've seen that the operation POS, when used with the primitive PRINT, will display the turtle's position with respect to the X and Y coordinates. The operations XCOR and YCOR used with PRINT will give us the information for the X and Y axes respectively. For example, we can now direct Logo to

```
?PRINT XCOR
-50
?PRINT YCOR
```

```
35
?□
```

and obtain the turtle coordinates. SETX, SETY, XCOR, and YCOR can be combined together in procedures, as shown below:

```
?TO STEP
>REPEAT 10 [SETX (XCOR+5) SETY (YCOR+5)]
>END

?STEP
```



Don't worry at this time why we placed the coordinate positions in parentheses ( ). We'll explain that point later in this text.

Practice using SETX, SETY, PRINT POS, XCOR, YCOR and HOME until you're comfortable moving (and tracking) the turtle on the screen. For fun, see if you can determine the number of steps the turtle can travel along the Y axis before it wraps around. Is the world of WRAP square? I.e., how many steps can the turtle take along the X axis before it wraps around?

In addition, practice using these primitives to make geometrical shapes similar to those we drew in Chapter 1. For example, let's think about what coordinates we need to use to draw a square with sides of 50 turtle steps in the exact center of the video screen.

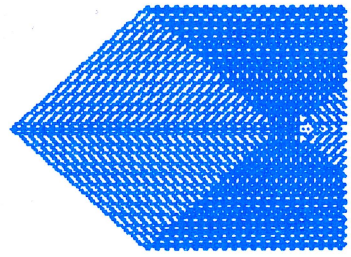
When you feel comfortable with the turtle universe of WRAP, proceed to the next section for a description of Logo's second screen mode—WINDOW.

2.3.2 WINDOW

The primitive WINDOW expands the size of the turtle's universe considerably. To see the effects of WINDOW, type in the following commands:

```
?WINDOW
?DIAMOND.PATTERN.HORIZONTAL
```

The following figure will appear after all thirty repetitions of DIAMOND.PATTERN.HORIZONTAL have taken place:



We will know when the procedure has ended as we'll see (if we're in SPLITSCREEN) the prompt and cursor reappear at the bottom of the screen. Note that in WINDOW, the turtle doesn't wrap around the screen, but continues to move to the right (and off!) the side of the screen.

Does the turtle's movement in WINDOW work the same way along the Y axis? To find out, clear the screen, and call the (vertical) DIAMOND.PATTERN procedure.

We can think of the video screen as a figurative window in this mode, with the turtle's universe extending out in both directions.

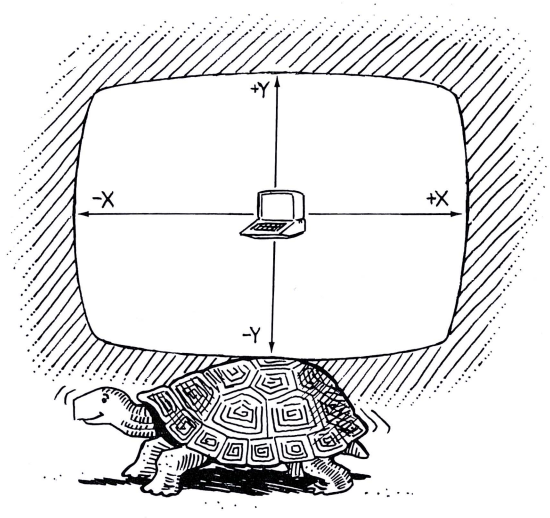


Fig. 2-11. Turtle universe in WINDOW.

What is the size of the window in this mode? How many steps can the turtle take in both the Y and X directions, before it disappears? You'll find, with experimentation, that the Y axis is about 240 steps and the X axis

about 280. Is this the same size as the screen in the WRAP mode before the turtle appeared on the other side?

You're right if you said "yes" to this question. Even though the turtle's universe is much larger in WINDOW, it's nice to know our own video screen has remained constant in size. We'll see that the same holds true for the final video mode, FENCE.

After we exceed the boundaries of the video screen, how far can the turtle travel along both the X and Y axes? What happens when you exceed these values? You'll find that the Euclidian universe of the turtle in WINDOW is actually quite large; it is approximately 40960 steps on the Y axis, and 37768 on the X as shown below.

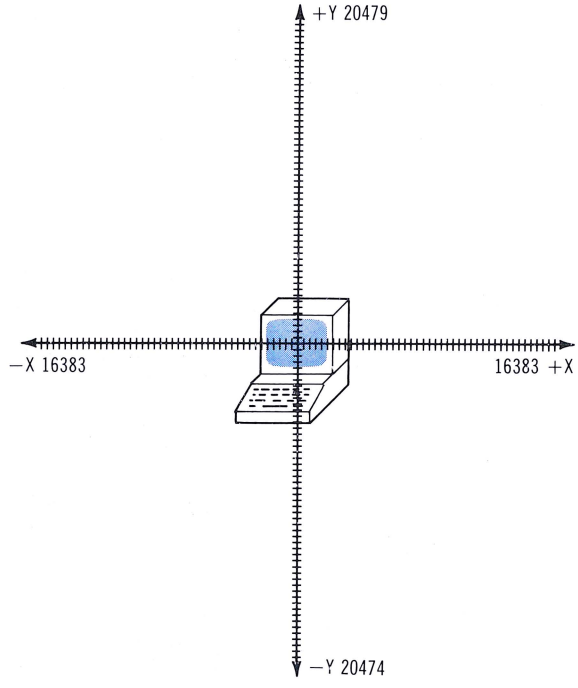


Fig. 2-12. Turtle boundaries in WINDOW.

What happens if we exceed these numbers as an input for a FORWARD command? If, for example, we are HOME and direct Logo to FORWARD 20500, it will respond with the message

```
TURTLE OUT OF BOUNDS
```

That is Logo's message that the turtle has exceeded the boundaries in WINDOW.

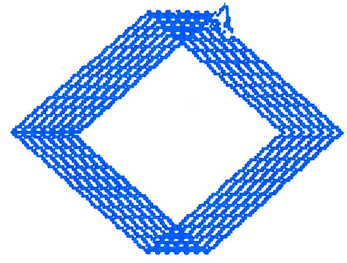
Earlier we asked whether the turtle's universe could ever be constricted. In our third and final screen mode, we'll find that the turtle's movement can, indeed, be limited to a much smaller area than in WINDOW.

2.3.3 FENCE

The third video mode is called FENCE. To see the effects of FENCE, clear the screen with CLEARSCREEN and type in the following:

```
?FENCE
?DIAMOND.PATTERN.HORIZONTAL
```

The screen will look like this:



The message on the bottom,

```
TURTLE OUT OF BOUNDS
```

suggests that FENCE limits the turtle's universe to the size of the video screen. The turtle can no longer WRAP around, and cannot move off the screen as in WINDOW. Is FENCE the same size as the video screen in WINDOW and WRAP? Try an experiment with SETX and SETY.

When you're finished exploring the three turtle universes, move on to the next section, where we add some color to the turtle cosmos!

Let's summarize our three modes and three levels with the diagram in Fig. 2-13.

2.4 Adding Color to the Turtle Tracks

In addition to altering the size and structure of the turtle's universe, Apple Logo allows us an array of colors to add to our graphics designs. This section is primarily written for those who have a color monitor or

color TV; the effects of the "coloring primitives" on a black and white, green, or amber screen are also interesting, but not as dramatic.

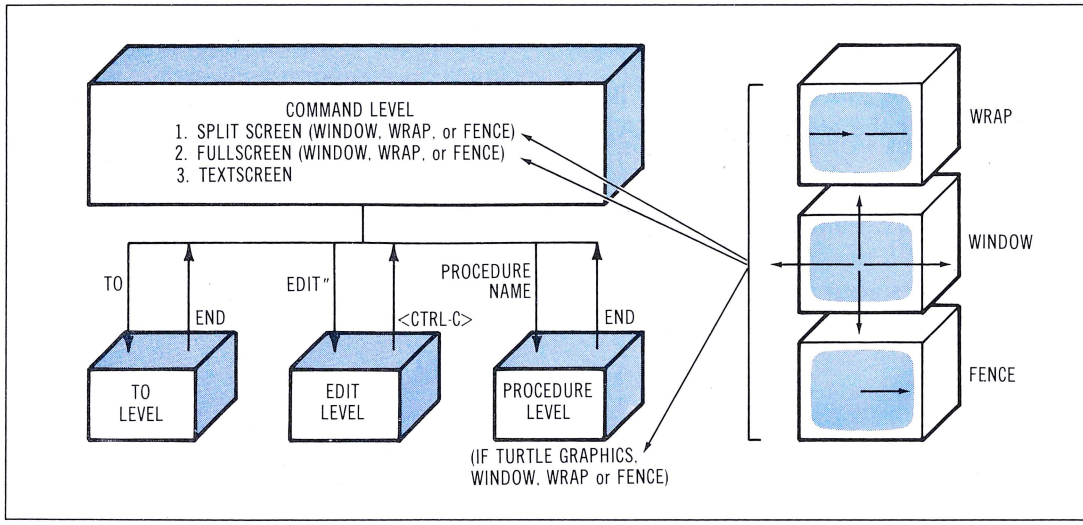


Fig. 2-13. Three modes and levels in Logo.

There are four primitives which deal with coloring our turtle graphics.

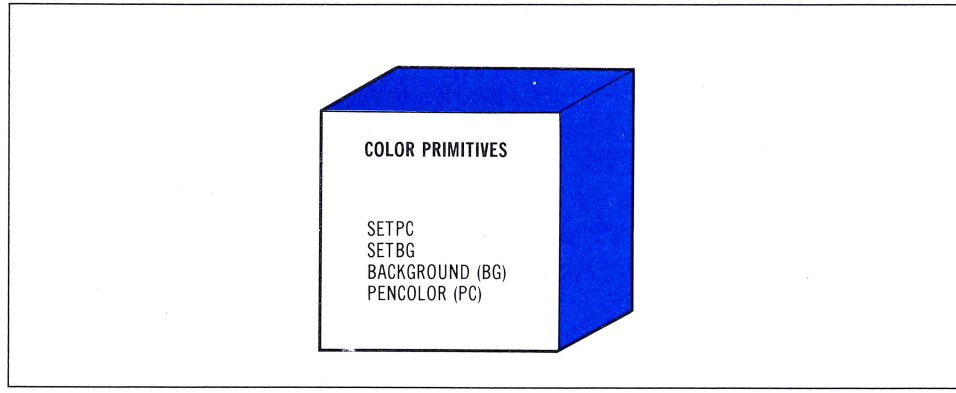


Fig. 2-14. Color primitives in Logo.

The primitives SETPC ("SET Pen Color") and SETBG ("SET Background Color") are used to change the turtle track or background screen color respectively. There is a choice of several colors, according to the following color code:

Color Number	Pen or Background Color
0	black
1	white
2	green
3	violet
4	orange
5	blue
6	black

To change the pen or background color, use SETPC or SETBG followed by one of these numbers. For example, to draw DIAMOND.PATTERN in orange turtle tracks on a green background, we'd need to type in the following commands, all at the Command Level:

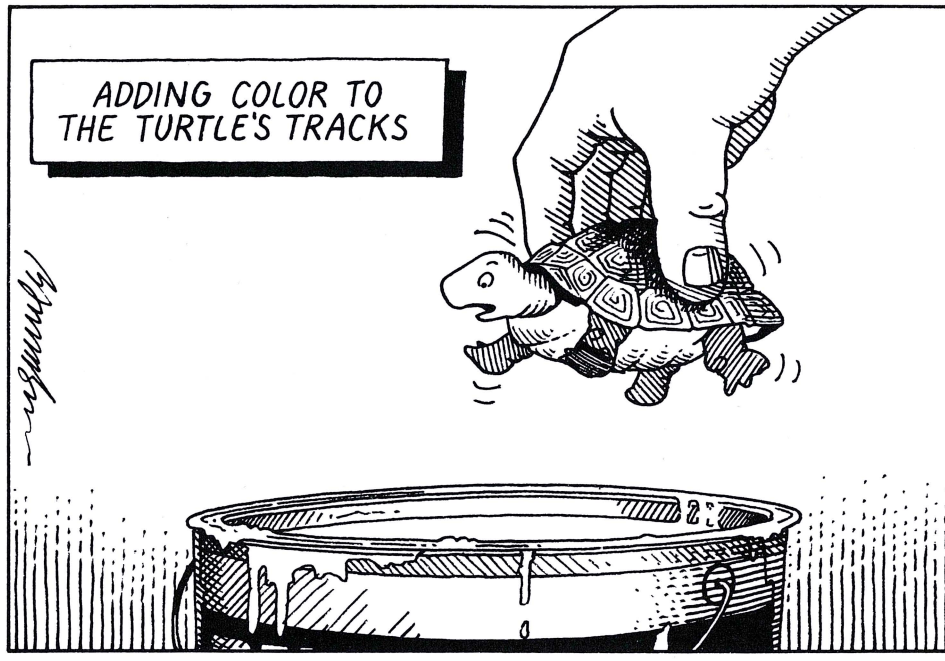
```
?SETBG 2
?SETPC 4
?WRAP                (or FENCE or WINDOW)
?FULLSCREEN          (if you like)
?DIAMOND.PATTERN
```

Try experimenting with other color combinations. What happens if you choose a pen color and a background color that are the same? Does the hue of different pen colors change somewhat depending on the background color? Try it and see.

The other two color primitives, BACKGROUND and PENCOLOR act like the primitive POS in our earlier section on coordinate locations. When used with the primitive PRINT, BACKGROUND and PENCOLOR direct Logo to display the current number of the background and pen color. If, for example, you're using blue pen color on a white background, Logo will display the current colors as follows:

```
?PRINT PENCOLOR
5
?PRINT BACKGROUND
1
?□
```

When you've finished using the different color combinations, restore the color mode to "normal." Note that there are two numbers (0 and 6) for the normal black color screen. Apple recommends you use 6 for the background, and 1 for the pen color under normal conditions.



2.5 TEXTSCREEN

We'll leave the world of turtle graphics for now and examine the final Logo screen mode, TEXTSCREEN. Whereas SPLITSCREEN and FULLSCREEN are primarily the home of the turtle, TEXTSCREEN takes us out of turtle graphics, into a "no draw" mode. Although we haven't called it such, when we first boot Logo into Apple, we're in TEXTSCREEN. We've gone from there into turtle graphics via SHOWTURTLE (or any graphics command) or FULLSCREEN.

To enter TEXTSCREEN, if you're in turtle graphics, use either of the two methods shown in Fig. 2-15.

You may either use the command TEXTSCREEN or the keyboard sequence CTRL-T. The screen will go blank, removing any graphics. The cursor and prompt will appear at the upper left hand corner of the screen.

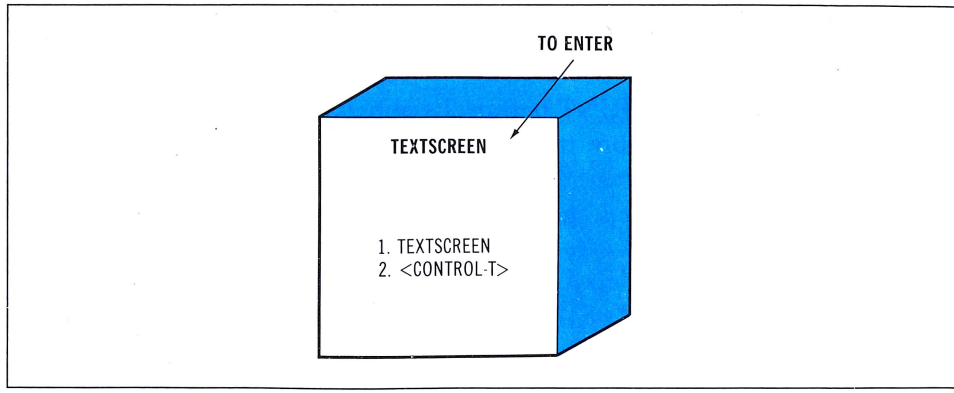


Fig. 2-15. Entering TEXTSCREEN.

2.5.1 Printing Words and Sentences

In TEXTSCREEN, the turtle assumes a "resting" stage and the screen is available for displaying words and sentences.

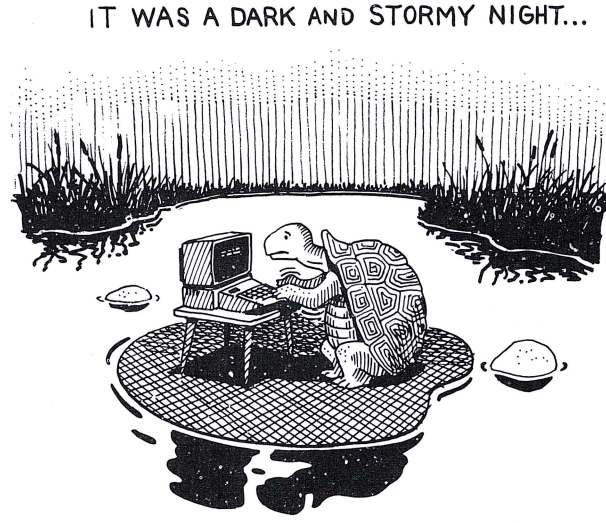


Fig. 2-16. Turtle writing.

Chapter 7 provides a complete description of how Logo deals with words and sentences. For now, however, the following guides will help. To display a single word, use the primitive PRINT followed by a quotation mark (") and the word. No quotation marks are needed after the word.

```
?PRINT "MARIJANE Note the space after PRINT
MARIJANE
```

We've used PRINT before. Its abbreviation is PR.

To display more than one word, or a sentence, use the PRINT (or PR) command followed by brackets enclosing what is to be displayed. For example, to have Logo display your full name, use

```
?PR [MY NAME IS MARIJANE PAULSEN]
MY NAME IS MARIJANE PAULSEN
```

PRINT, like other Logo primitives, can be used to write procedures. For example:

```
?TO GERTRUDE.STEIN
>REPEAT 10 [PRINT [A ROSE IS]]
>END
```

The characters inside the brackets are called a "list." When we direct Logo to GERTRUDE.STEIN, it will respond by printing the list A ROSE IS ten times.

What happens if we write a very long sentence? If the characters in the list exceed the number of characters that can be fitted on a line, Logo will automatically do a "carriage return" like a typewriter. It will place an exclamation mark (!) at the spot where the carriage return occurs. For example, a long sentence ("He who hesitates is not only lost, but miles from the nearest exit") will be displayed like this:

```
?PRINT [HE WHO HESITATES IS NOT ONLY LOS!
T, BUT MILES FROM THE NEAREST EXIT.]
HE WHO HESITATES IS NOT ONLY LOST, BUT M!
ILES FROM THE NEAREST EXIT.
```

For lists in brackets that exceed the length of your video screen line the exclamation point functions as a hyphen, or as an indication that there's more to come. Unfortunately, Logo's knowledge of English does not extend to syllables, so your words may be broken up in some s!trange places . . .

2.5.2 Logo as a Calculator

In addition to graphics and text capabilities, Logo is a fine "number cruncher." To have Logo perform as a calculator, we use the primitive

PRINT along with the familiar arithmetic symbols "+" and "−" for addition and subtraction. To multiply and divide, use "*" and "/". These functions are summarized below.

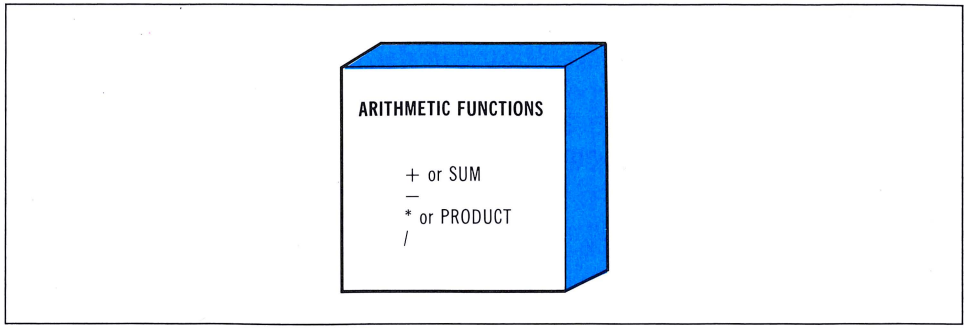


Fig. 2-17. Arithmetic functions.

Here are some examples.

```
?PRINT 3 + 6
9
```
Note no quotes

```
?PRINT 3+6
9
```
Note that spaces before and after the arithmetic sign aren't needed

```
?PRINT 2 * 4 + 6
14
```
Multiplication is done before addition.

```
?PRINT 2 * (4 + 6)
20
```
We use parentheses to first add 4 to 6, then multiply that by 2.

```
?PRINT 2 * 4 + 6 + 1
15

?PRINT 2 * (4 + 6 - 1)
18
```

The rules of precedent apply in Logo as in the rest of the world, i.e., multiplication and division are done before addition and subtraction. You can overrule that rule by grouping numbers in parentheses, as we did in the last example.

We can also use arithmetic operations in procedures, as in the following example:

```
?TO NINES
>PRINT [NINE TIMES SIX IS . . . ]
>PRINT 9 * 6
>END
```

When we run the program NINES, we'll see the following:

```
?NINES
NINE TIMES SIX IS . . .
54
```

What would happen if on the third line of the procedure we enclosed "9 * 6" in brackets? Here is a test.

```
?PRINT 9 * 6
54

?PRINT [9 * 6]
9 * 6
```

In the first example, Logo performed the multiplication. In the second example, Logo displayed the characters inside the brackets verbatim.

Later chapters will explore numbers, words and lists in more detail.

2.6 Summary

You've learned quite a few "turtle management" commands and operations, as listed below:

1. Entering the Three Screen Modes in Logo

FULLSCREEN	CTRL-L
TEXTSCREEN	CTRL-T
SPLITSCREEN	CTRL-S or any turtle graphics command

2. Controlling the Turtle's Boundaries

WINDOW	Gives turtle large area outside screen to play

FENCE	If turtle leaves screen, error message is given
WRAP	Normal condition, turtle wraps to other side of screen

3. XY Coordinate System

XCOR	Gives the turtle's present X-coordinate.
YCOR	Gives the turtle's present Y-coordinate.
SETX 50	Moves the turtle to X value of 50.
SETY 50	Moves the turtle to Y value of 50.
HOME	Moves the turtle to X = 0, Y = 0.
POS	Gives turtle's present XY coordinates.

4. Adding Colors

PENCOLOR PC	Gives current pen color.
BACKGROUND BG	Gives current background color.
SETPC 1	Sets pen color, 1 in this case.
SETBG 6	Sets background color, 6 in this case.

0	Black	Normal
1	White	BG 6
2	Green	PC 1
3	Violet	
4	Orange	
5	Blue	
6	Black	

2.7 Review Questions and Exercises

Review Questions

1. When you first boot Logo, what screen mode are you in?
2. a. What is the "normal" screen mode when using the turtle?
   b. What command(s) puts you in this mode?
3. What can you do in the Command mode that you can't do at the TO level?
4. What happens in the Procedure level? How do you get in it? How can you get out of it?
5. Which of the primitives, FENCE, WINDOW, or WRAP applies under "normal" conditions? Which primitive lets the turtle go off the screen?

6. What are the X,Y coordinates of the center of the turtle's screen?
7. The "SET" commands (SETX, SETY, SETPC, and SETBG) are used to give the turtle specific values. What corresponding primitives output the turtle's present values?
8. When using simple Logo arithmetic primitives, what is the result of these commands?

```
PRINT 4+5*6
PRINT 4-6
PRINT 8/2-3
```

9. Will the following commands work?

```
FD 3*6
FD 3-6
```

Exercises at the Terminal

1. Place a large pattern on the screen, like DIAMOND.PATTERN.
   a. Practice using CTRL-S, CTRL-T, and CTRL-L.
   b. Practice using the editor to change some values. When you leave the editor, what mode are you in?
2. Try the following procedure to "walk" the turtle.

```
TO W
FD 5
PRINT POS
END
```

3. Use the editor to change the name of the procedure DIAMOND.SHAPE.HORIZONTAL to D.
   a. Try to run the procedure D. Does it work?
   b. Try larger values of inputs to FD in the procedure D.
   c. Now type WINDOW and try some more large values of inputs to FD.
4. Can you create a procedure that draws a triangle
   a. using just FD and RT?
   b. using just SETX and SETY?
5. If you have a color display, experiment with the procedure D (see exercise 3) by adding color to the pen and background.

Answers to Review Questions

(1) TEXTSCREEN. (2) SPLITSCREEN is the normal turtle graphics screen mode. The commands that put you into SPLITSCREEN include SHOWTURTLE or ST; any turtle graphics primitive, like FD, BK, or RT; [CTRL-S]; or SPLITSCREEN. (3) At the Command Level, you can call different screen modes, and direct Logo to place you in the TO, Edit, and Procedure Levels. (4) Procedures and programs are executed in the Procedure Level. You enter the procedure level by typing the name of the procedure or program you want to run; you get out of it when the procedure or program has ENDed. (5) WRAP is the "normal" condition. The primitive WINDOW lets the turtle go off the screen. (6) At the center of the turtle's screen, X = 0 and Y = 0. (7) PRINT XCOR, PRINT YCOR, PRINT PENCOLOR, and PRINT BACKGROUND. (8) 34, −2, 1. (9) Yes, the first would direct the turtle to go FORWARD 18 steps; the second to go FORWARD −3 steps.

CHAPTER 3

THE STRUCTURE OF LOGO

CONCEPTS:

Logo's underlying structure
commands and operations
types of inputs
using variables as inputs to procedures
formal definitions of primitives

PRIMITIVES:

ROUND, PRODUCT, SUM

"When ideas fail, words come in very handy." (Goethe 1749-1832)

3.0 Introduction

In Chapter 1 we focused on the micro side of Logo, i.e. how to get the language up and running and how to use commands to call and move the turtle. We also saw that commands could be grouped together to teach Logo new words, called procedures. In Chapter 2, we took a more macro or global view, and developed an understanding of Logo's level of command and screen modes.

So far, you might say we've looked at the trees and the forest. In this chapter, we'll focus on the "ecology" of the system. We'll see how the language of Logo is put together. We'll see that Logo is considered to be a "high level" language, since its structure is close to our "natural" language, English.

We'll see that Logo's syntax allows us to write procedures with "variables." That is, we can write generic procedures which will accept different input values. We'll be able, for example, to write a procedure that can be used to draw a square of 10, 25, 66, 70, or any variable turtle step size we want.

We then conclude the chapter by taking a closer look at Logo primitives and giving them "formal definitions." These formal definitions provide a concise summary of what each primitive does. The formal definitions also serve to strengthen our understanding of Logo syntax.

We'll begin the chapter with an overview of Logo's structure.

3.1 An Overview of the Logo Language

All languages, from English to French to Logo, have a fundamental structure. For example, English, for all its rich vocabulary and ability to communicate abstract thought, has a very simple underlying structure: all sentences have a subject and a predicate or verb. Likewise, we search for and recognize underlying structures in other situations. All the wooden buildings that are on our street have the same pattern of an internal frame support over a concrete foundation. Of the dozens of car models you can buy today, all have a chassis and an engine. All vertebrate animals have an internal support system hooked to a backbone.

Logo also has an internal skeleton. No matter how long and complex a Logo program may look, it has a discernible underlying structure. Each and every line of a Logo program has one of two forms as shown in Fig. 3-1 below:

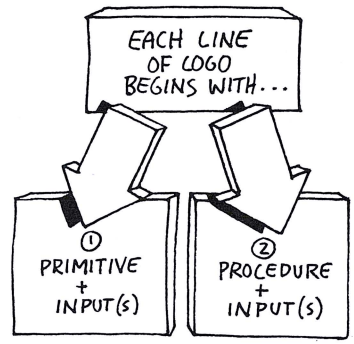


Fig. 3-1. The structure of LOGO.

Although we did not define it as such in Chapters 1 and 2, all of our programming so far has been based on this structure. For example, in Chapter 1, we used the primitive FORWARD followed by an input (such as 50) to move the turtle 50 spaces. The primitive FORWARD requires an input in order to operate. Here are some examples of primitives and their inputs:

PRIMITIVE	INPUT
BACK	25
RIGHT	70
PRINT	''WOW
REPEAT	3[FORWARD 20 RIGHT 120]
PENUP	

The first three examples each have one input following them; the fourth, the REPEAT command has two inputs—the number "3" and the list [FORWARD 20 RIGHT 120]. The primitive PENUP works without an input. It is important for us to know the number of inputs a primitive or procedure requires. Fortunately for us, Logo is a forgiving language. It tells us when and where we've made an error concerning inputs. If we do not give a primitive or procedure enough inputs to carry out its work, Logo will respond with an appropriate error message indicating the source of the problem. If you type

```
?FORWARD
```

and press RETURN without an input, Logo will respond with the message

```
NOT ENOUGH INPUTS TO FORWARD
```

If we err on the other side, and exceed the correct number of inputs for a command, Logo will also gently complain. For example, if you direct Logo to

```
?FORWARD 30 50
```

it will first move 30 turtle spaces, and then print

```
I DON'T KNOW WHAT TO DO WITH 50
```

Logo accepts the first input, the number "30." We say that the input, "30," is "fed" to the primitive FORWARD. Each and every input must be fed to a primitive or procedure. FORWARD will only accept one input; in the example above, the input "30" was fed to FORWARD. The error message occurred since there was no primitive or procedure to feed the input "50" to.

So far, we've seen that Logo's internal structure is based on two types of information we give it: primitives or procedures, and their inputs.

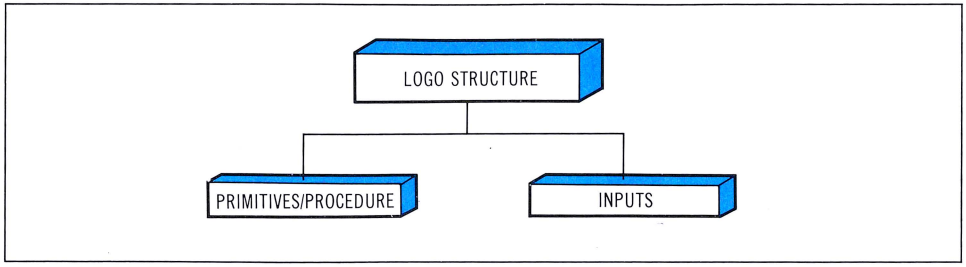


Fig. 3-2. Logo structure: The simplest view.

The purpose of inputs is to pass information to the primitive or procedure; the purpose of the primitive or procedure is to get Logo to do something.

Now let's look at primitives and procedures in more detail.

3.1.1 Two Kinds of Logo Primitives and Procedures

We saw in the last section that one way we could classify primitives would be based on the number of inputs they take. There is, however, another important way of differentiating between Logo's action words. Essentially, all Logo primitives or procedures can be divided into those that are commands and those that are operations. The big difference between them is that operations produce an "output," while commands do not. And, the output from an operation must be used as an input to a primitive or procedure.

Most of the Logo primitives we've seen thus far have been commands. The following are examples of Logo commands:

```
?PRINT "HELLO
?FORWARD 30
?REPEAT 4 [FORWARD 90 RT 90]
```

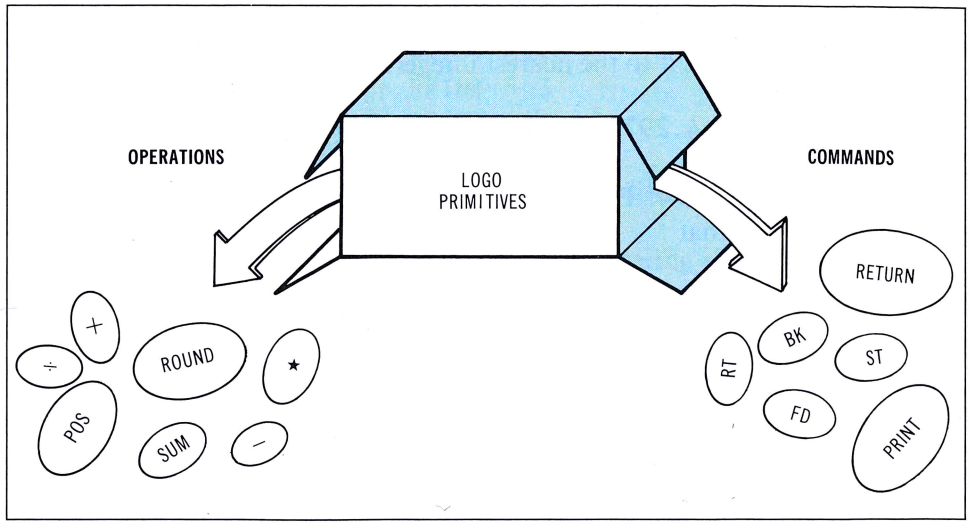


Fig. 3-3. Commands vs. operations.

We see that primitives like PRINT, FORWARD, and REPEAT are like verbs requiring inputs to have them perform as we wish.

In Chapter 2, we worked with the arithmetic "operations" add, subtract, multiply, and divide. The symbols +, −, *, and / are Logo operations. They are different from commands in that they produce an "output" which must be passed on to a Logo primitive or procedure. For example, if you type

```
?15 + 32
```

and press RETURN, Logo will respond by displaying

```
I DON'T KNOW WHAT TO DO WITH 47.
```

Logo was able to do the arithmetic operation, but the result of the operation it performed needs to be given to a LOGO primitive. The way we do this is to "feed" the result of an operation to a primitive like PRINT. Thus, if you type

```
?PRINT 15 + 32
```

Logo will do the arithmetic operation, and then pass the number 47 to PRINT. We call the number 47 an "output." Outputs become inputs to primitives such as PRINT.

Another Logo operation is ROUND. ROUND takes a (decimal) input and rounds it to the nearest integer. For example,

```
ROUND 4.297
```

will output 4. But when you ask Logo to ROUND 4.297, it will patiently explain that

```
I DON'T KNOW WHAT TO DO WITH 4.
```

The output 4 must be fed to a primitive like PRINT. Thus,

```
?PRINT ROUND 4.297
```

will display 4. You might think of the sequence as occurring in the following order as shown in Fig. 3-4 below.

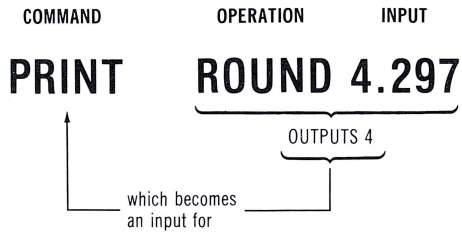


Fig. 3-4. Operations produce outputs that become inputs.

Logo first ROUNDS 4.297 to 4; that output becomes an input to PRINT. Our final example is the Logo operation PRODUCT. PRODUCT takes two inputs, and outputs their product. For example,

```
PRODUCT 3 4
```

outputs 12. As with the operations "+" and ROUND, this output must be fed to a primitive such as PRINT. Thus, we would type

```
?PRINT PRODUCT 3 4
```

to display 12.

We might mention now that the arithmetic symbols have word equivalents for their operations. Logo would happily accept either of the following:

```
?PRINT PRODUCT 9 10 or PRINT 9*10
?PRINT SUM 6 3      or PRINT 6+3
?PRINT SUM 5 3      or PRINT 5-3
?PRINT QUOTIENT 6 2 or PRINT 6/2
```

These and other arithmetic operations are discussed in more detail in Chapter 10.

Before we leave this area, you might be wondering if operations can output their values to other operations. Yes, they can. Here is a complex example:

```
?PRINT PRODUCT SUM 2 3 SUM 4 5
45
```

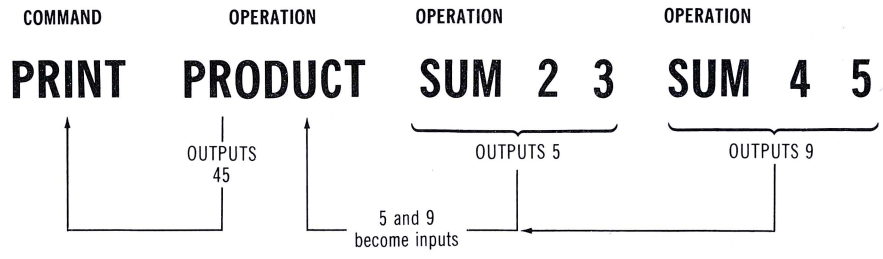


Fig. 3-5. Operations output their values to other operations.

We'll summarize our classification of Logo primitives and procedures by expanding the chart we began earlier:

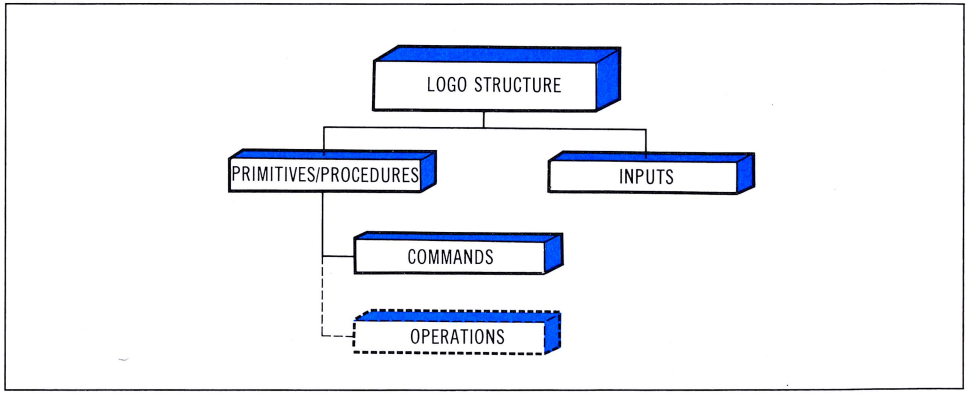


Fig. 3-6. Logo structure showing operations and commands.

We saw in this section that there are two kinds of primitives: commands and operations. We've seen that commands direct Logo to perform specific tasks. Each and every line of a Logo procedure must begin with a command.

Operations output their value as an input for a command or another operation. We've put dotted lines around the operations box since later in this chapter, we'll argue that it may make better sense to put operations on the input side of Logo's structure! For now, let's leave the primitives/procedures sides of Logo as it is, and examine the variety and form of inputs that primitives can take.

3.1.2 Five Kinds of Inputs

We've seen that inputs are important because they pass the information to a command. Inputs come in five "flavors"; they may be as follows:

Types of Inputs
(1) numbers
(2) words
(3) lists
(4) dotted words
(5) operations

Although we've used the first three types of inputs in writing Logo procedures, let's review them with some examples:

(1) number inputs

```
FD 60
RT 90
PR 50
BK 20
```

(2) word inputs

```
PRINT "HELLO
PRINT "WOW
```

(3) list inputs

```
PRINT [THE FLIP SIDE . . . ]
PRINT [ALSO HAS A FLIP SIDE.]
```

Before we get a taste of these inputs, let's incorporate them in our Logo structure diagram.

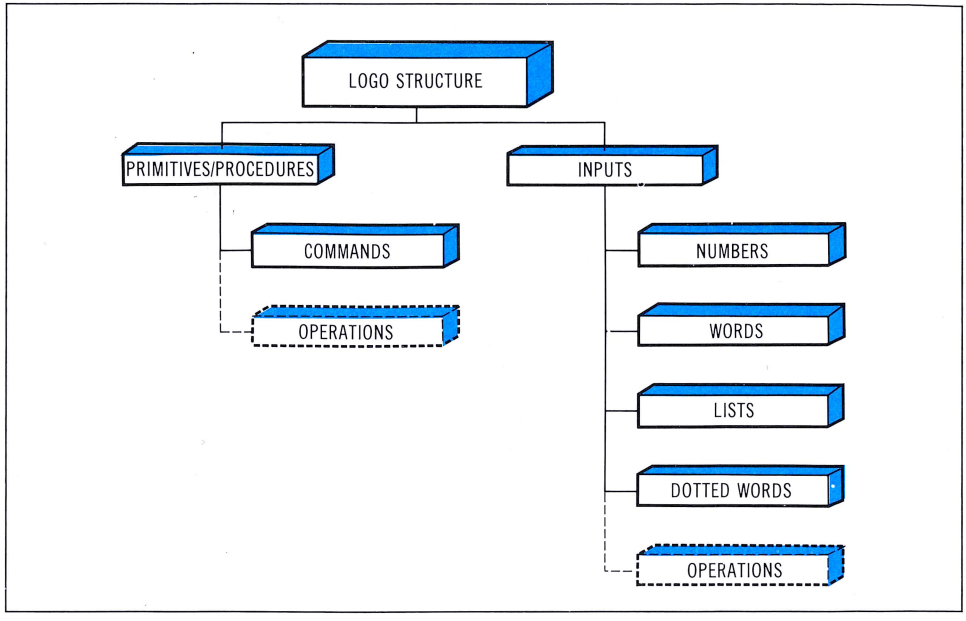


Fig. 3-7. Logo structure showing inputs.

Numbers, words, and lists are called "data objects" in Logo. Data objects are the "basic" inputs for Logo commands and operations. We've had a brief introduction to data objects in Chapters 1 and 2 and we'll examine them more closely in Chapter 7.

In addition to numbers, words, and lists, Logo also accepts dotted words and operations as inputs. These two input types are described below.

(4) dotted words as inputs

Dotted words are "variables." Variables increase our programming flexibility considerably, as the following example (and the rest of this book!) will illustrate.

In Chapter 1, we wrote a procedure which directed the Logo turtle to draw a square as follows:

```
?TO EASY.SQUARE
>REPEAT 4[FD 50 RT 90]
>END
```

Suppose we wanted to draw squares of varying sizes. We could edit our EASY.SQUARE program each time and change the input to FORWARD from 50 to 60 to 30 and so on. Another way would be to write a "generic" procedure for EASY.SQUAREing, and indicate to Logo the size we want by using inputs. We begin our generic procedure by indicating on the first program line that we are creating a variable input to the procedure:

```
?TO VARY.SQUARE :SIZE
```
The TO line contains the variable input "SIZE"

This line informs Logo that we're writing a procedure with a variable input. The colon before SIZE is read as "dots." It tells us that SIZE is the variable name. Note that there is no space between the colon and the name SIZE. In the body of the procedure, we substitute the variable :SIZE for our previous number input to the FORWARD command, as follows:

```
?TO VARY.SQUARE :SIZE
>REPEAT 4 [FD :SIZE RT 90]
>END
```
We place :SIZE in the body of our procedure

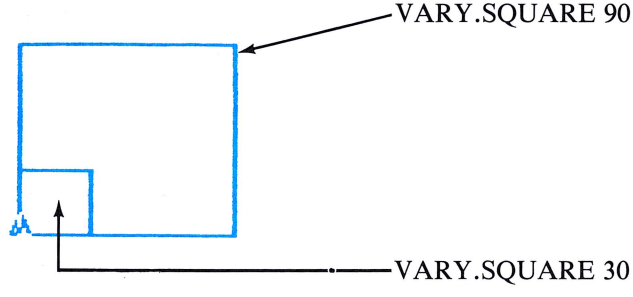
To run the program, provide a number input after the procedure name. Thus, type in

```
?VARY.SQUARE 30
```

Logo responds by drawing a square with sides of 30 turtle steps. If you now direct Logo to

```
?VARY.SQUARE 90
```

it will draw a square with sides of 90 turtle steps.



We say that the value of :SIZE in VARY.SQUARE is "passed" on to FORWARD when we run the procedure. Thus, VARY.SQUARE 30 passes the number 30 on to FORWARD since both use the same input name, :SIZE.

We'll return to variables later in this chapter, and also see how we can write procedures with more than one variable. For now, remember that dotted words or variables carry a value, which can be a number as we saw in VARY.SQUARE :SIZE. Dotted words can also be words, lists, other dotted words, or operations. Variables have a great deal of variety!

(5) operations as inputs

Our final type of input is our new friend, the operation. For example, in

```
?PRINT SUM 3 4
?PRINT ROUND 99.82
?PRINT PRODUCT 6 9
```

SUM, ROUND, and PRODUCT are operations. We've seen that these operations produce outputs. These outputs, which must be used as inputs, will be in the form of a number, word, list, a dotted word, or even another operation.

We can make one more generalization that helps us understand Logo's structure. All single words (or unbroken "strings" of characters) in Logo are one of three kinds. They can be any of the following:

quoted words like "HI
dotted words like :SIZE
unmarked words like PRINT

Quoted words are one of Logo's three basic data objects, called "words" as we discuss further in Chapter 7. Dotted words provide the value for variables as we discuss in Chapter 9. Unmarked words are interpreted by Logo to be either primitives or procedures. Thus, if Logo is given an unmarked word like SMILE in the form below:

```
?SMILE
```

it checks to see if it's a primitive. It's not.

Then it checks to see if it's a procedure. If not, we get the error message

```
I DON'T KNOW HOW TO SMILE
```

This generalization applies to all single words that are not contained in lists (words contained in lists are discussed in Chapter 7).

Let's add this information along with a summary of Logo inputs to the diagram is Fig. 3-8.

And that, fellow programmers, is the structure of Logo, often called its syntax. All Logo programs, no matter how simple or complex, are based on this same skeletal framework. You'll also see why we argued earlier that operations may best be placed on the input side of Logo. Although they're primitives or procedures, they are always used as inputs. We have decided to keep them on both sides of the diagram.

Let's summarize Logo's syntax by looking at some examples of Logo expressions and identify the elements they contain:

EXPRESSION	STRUCTURE
`CLEARSCREEN`	command
`PRINT 40`	command plus one input, a number
`REPEAT 3 [PR "HELLO]`	command plus two inputs—a number and a list
`END`	command
`EDIT "SQUARE`	command plus one input—a word
`FORWARD :SIZE`	command plus one input—a dotted word (the value must eventually be a number)
`PRINT SUM 40 50`	command plus one input—an operation (which takes two inputs)
`BACK SUM 40 50`	command plus one input—an operation (which takes two inputs)

In the last example, the turtle would move BACK 90; the operation SUM would output 90 to become an input for BACK.

This "ecological" view of Logo demonstrates its basic structural simplicity. It also provides a hint of Logo's powerful data handling capabilities. When you realize that an operation may be dozens of lines long and that it can pass information to another operation, and so forth, you can quickly reach a level of program complexity which is very difficult to follow. Although complexity in itself is most undesirable in programming practice, this hierarchical characteristic of operations and dotted words along with other Logo characteristics provide the degree of sophistication necessary for

artificial intelligence (AI) programming similar to that done with the LISP language.

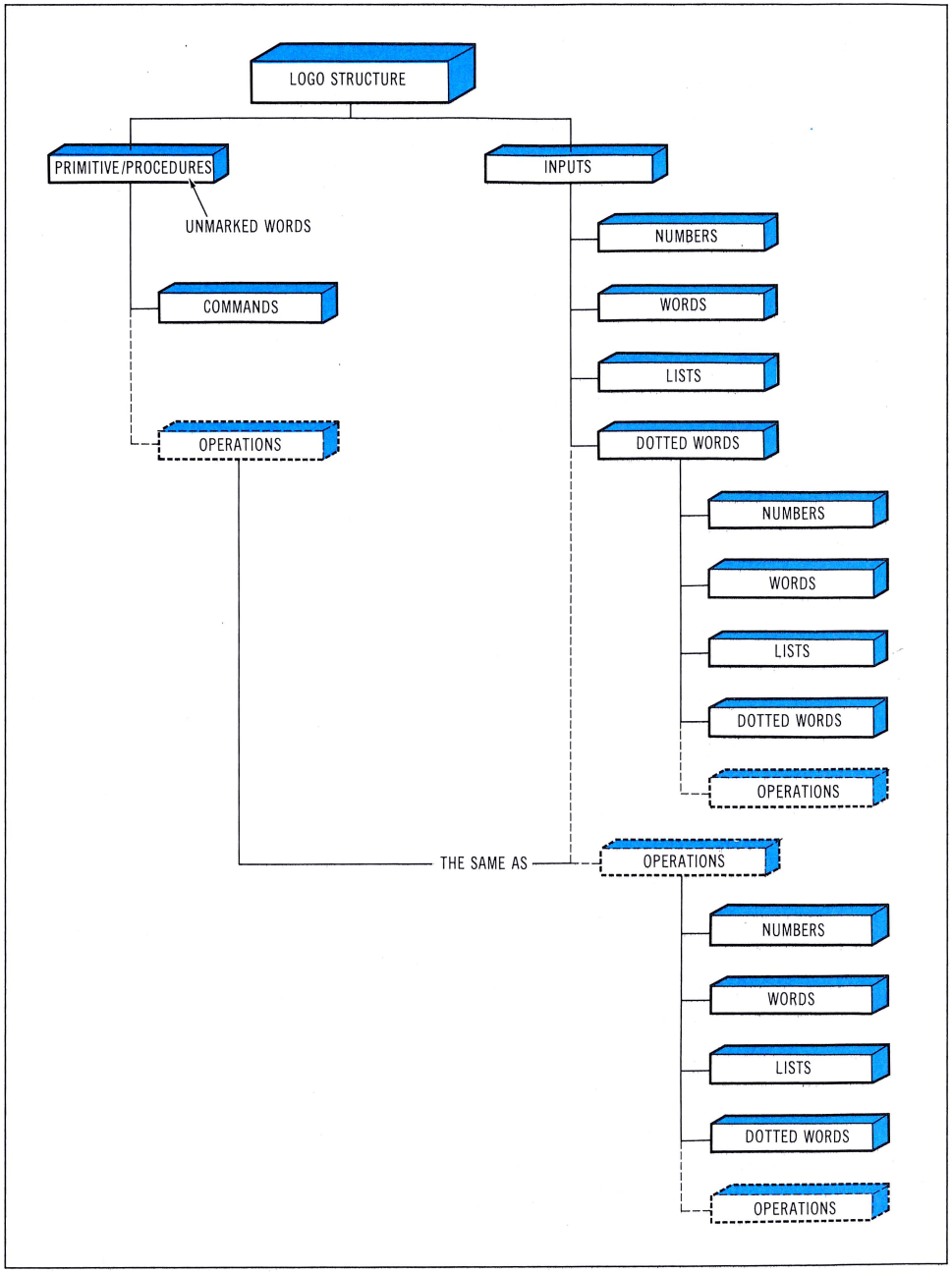


Fig. 3-8. Full Logo structure.

3.2 Variables as Inputs to Procedures

Earlier, we saw that dotted words, or variables, are one type of input to Logo procedures. We used a variable, :SIZE, to write a generic program to draw a square, and then assigned a value to the variable when we called the procedure. In this section, we'll explore variables further; we'll see how they're named and how to write procedures with multiple variables.

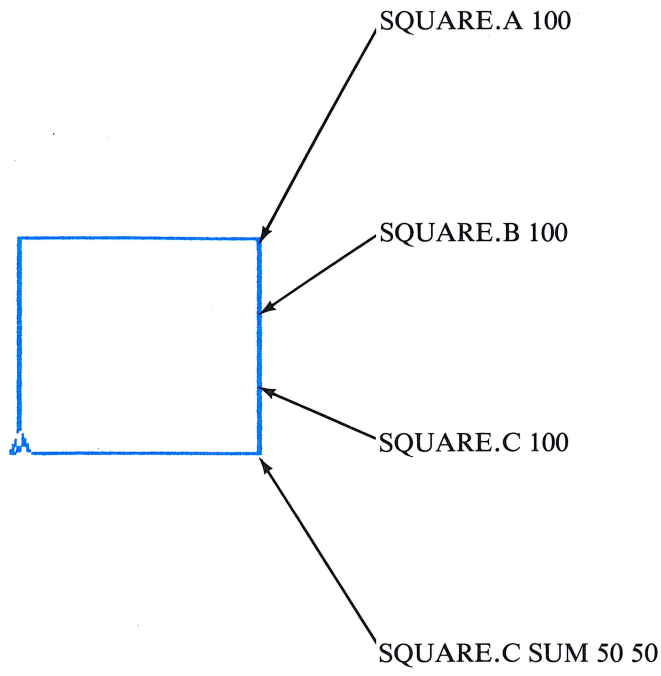
3.2.1 Names of Inputs

What's in a name? Just as garlic by any other name still smells the same, variables by any other name still work the same. For example, consider these three procedures:

```
?TO SQUARE.A :SIZE
>REPEAT 4 [FD :SIZE RT 90]
>END

?TO SQUARE.B :X
>REPEAT 4 [FD :X RT 90]
>END

?TO SQUARE.C :RAISIN
>REPEAT 4 [FD :RAISIN RT 90]
>END
```

All three programs will work exactly the same way, i.e. they will all draw squares of whatever size we input for the dotted word. The name of the input (SIZE, X, or RAISIN) is not important to Logo. These dotted words are called "dummy" variables. They are placeholders, a way to pass information on to FORWARD. What is important is the value of the input we assign to the variable. We could run any of the programs as follows:

```
?SQUARE.A 100

?SQUARE.B 100

?SQUARE.C 100

?SQUARE.C SUM 50 50
```



All of the above procedures would produce the exact same drawing: a square with sides of 100 turtle spaces. The reason that they all do the same thing is that the input value is the same.

Dotted words are used to hold information. The names we give the dotted words are to help human programmers remember what kind of value to give the input. Logo doesn't care if we call the length of a four-sided figure "raisin," but "size" would make better sense to us. An important suggestion for good programming is to choose meaningful names for dotted word inputs.

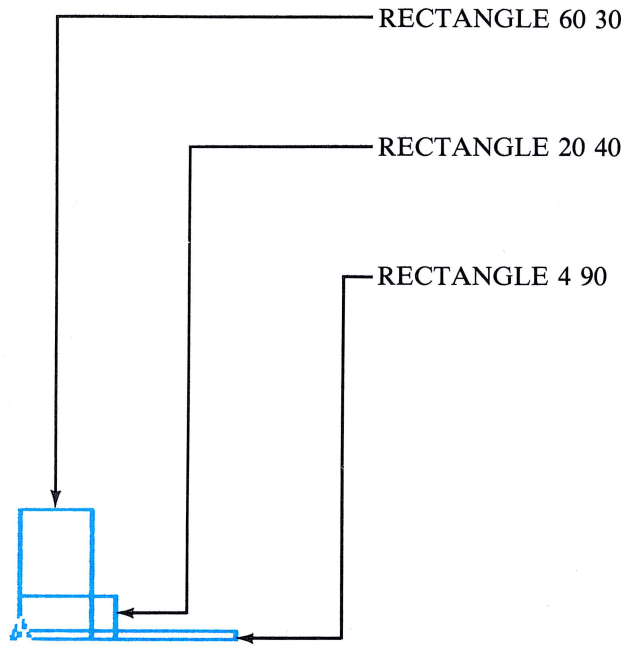
3.2.2 Writing Procedures with Multiple Inputs

Suppose we move from drawing squares to rectangles. Our inputs might now include the height and length of a four-sided figure. Thus, we might write a rectangle program as follows:

```
?TO RECTANGLE :HEIGHT :LENGTH   List the inputs :HEIGHT and :LENGTH
>FD :HEIGHT RT 90               The value of HEIGHT is passed here
>FD :LENGTH RT 90               The value of LENGTH is also passed on
>FD :HEIGHT RT 90
```

```
>FD :LENGTH RT 90
>END
```

We establish, on the TO line, that there are two inputs, HEIGHT and LENGTH. When we run the procedure, then, we'll need to provide Logo two values for these inputs:



Let's try another example. In Chapter 1, we suggested that you write a procedure called TRIANGLE. Now let's write the procedure using variables:

```
?TO TRIANGLE :SIZE
>RT 30
>REPEAT 3 [FD :SIZE RT 120]
>LT 30
>END
```

We could make an interesting design as we did in Chapter 1 by taking the

triangle shape and rotating it through 360 degrees to make a drawing called ROTATE.TRIANGLE:

```
TO ROTATE.TRIANGLE
>REPEAT 20 [TRIANGLE 50 RT 18]
>END
```

However, think of how much more powerful a generic procedure would be. Let's write a procedure to vary the number of rotations. We can vary the number of times the triangle shape is drawn, and the number of degrees we turn RIGHT. The procedure, TO VARY.ROTATE, might look like this:

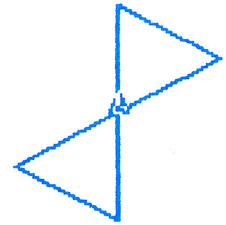
```
?TO VARY.ROTATE :NUMBER :ANGLE
>REPEAT :NUMBER [TRIANGLE 50 RT :ANGLE]
>END
```

Add the variables :NUMBER and :ANGLE as inputs here

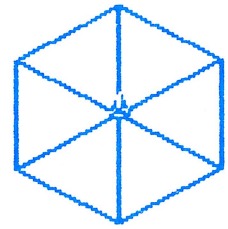
Indicate where the values for :NUMBER and :ANGLE are to be placed

You might be wondering if we could add the variable :SIZE to the inputs to VARY.ROTATE. We didn't do it, but you could. Just replace 50 by :SIZE and add :SIZE to the TO (the title) line.

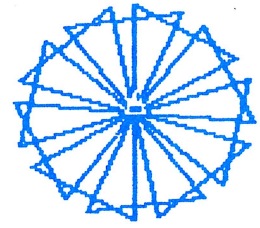
To run this procedure, designate the values for :NUMBER and :ANGLE, as shown in the following examples:



```
?VARY.ROTATE 2 180
```



```
?VARY.ROTATE 6 60
```



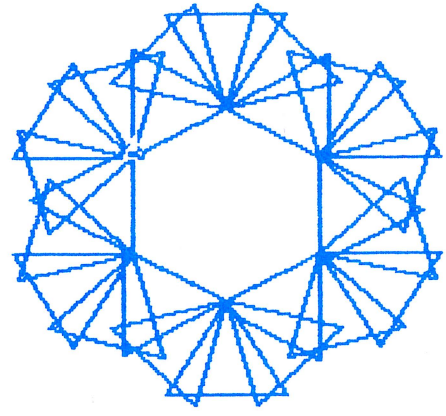
```
?VARY.ROTATE 10 36
```

Can we now write a procedure that includes VARY.ROTATE as a subprocedure? Can procedures call other procedures? Can they use inputs too? But of course! Logo has a very uniform structure for building up complex programs from simple ones. Consider the following program:

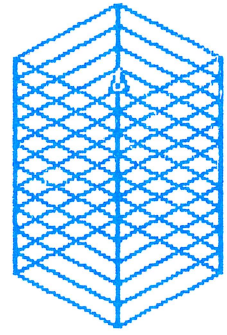
```
?TO DESIGN :NUMBER :ANGLE :DISTANCE :TURN
>REPEAT 6 [VARY.ROTATE :NUMBER :ANGLE FD :DISTANCE RT
 :TURN]
>END
```

DESIGN contains VARY.ROTATE, and two more variables: DISTANCE and TURN. The idea is to draw a shape made up of rotating triangles (with the variables NUMBER and ANGLE), move FORWARD a (variable) distance, and then turn RIGHT a (variable) angle.

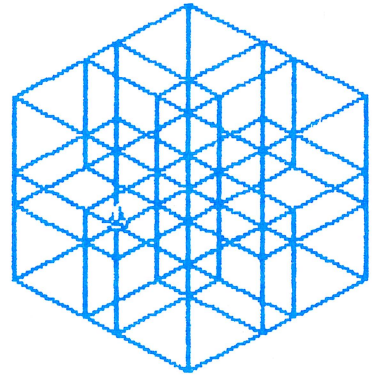
When we run DESIGN, we need to supply it with four number inputs. Some of our favorites follow.



```
?DESIGN 6 60 35 60
```



```
?DESIGN 4 315 45 120
```



```
?DESIGN 6 300 10 360
```

Changing the inputs to a neat procedure like this is a lot of fun. You can get all sorts of interesting designs. In the next chapter, we will describe how to save procedures that we want to keep permanently to disk. We will also describe how to obtain a "hard copy" if you have a printer.

3.3 Giving Formal Definitions to Primitives

Learning a computer language is a lot like learning English. We learn our human language by developing an understanding of both its structure and

its vocabulary. Then we spend considerable time learning to put the (correct) words within the (correct) structure to produce meaningful speech.

In learning Logo, we use the same skills: we need to understand its structure, know its vocabulary, and practice using the right information and words in the best places to produce programs. Thus far, this chapter has focused on the structure of Logo. Before we begin writing programs, we need to give attention to Logo's vocabulary, its primitives. We'll begin the process by developing formal definitions for primitives. These formal definitions will include information on the number of inputs the primitive expects, as well as the purpose, abbreviation, and expected output of the primitive.

Let's demonstrate our formal definitions with the often used primitive FORWARD. The formal definition of FORWARD is as follows:

FORWARD	**(takes one input)**
Purpose	Draws a line of specified distance forward on the graphics screen
Abbrev	FD
Inputs	Real number input; range determined by FENCE, WINDOW, or WRAP
Outputs	None. (FORWARD is a command)

Let's see how the knowledge of FORWARD, as defined, will keep us from programming "bugs." The following are procedure lines containing FORWARD that would work:

Correct Procedure Lines Containing FORWARD

```
FD 30          (FD takes one number input)
FD SUM 20 30   (output of SUM would be a number input to FD)
FD :SIZE       (If the variable: SIZE is assigned a number the line is okay)
```

The following lines would be incorrect procedure lines using FD for the reasons given:

Incorrect Procedure Lines Containing FORWARD

```
FD "HELLO      (Input from FD must be a number)
FD 40 to 50    (Input for FD must be one number)
FD [FAR AWAY]  (Input for FD must be a number)
```

```
PRINT FD 30     (FORWARD gives no outputs to commands like PRINT)
FD 30 90        (FORWARD takes only 1 input)
```

Let's define a few more primitives that we've used so far. In Chapter 1, we used the primitive PENUP. When we type in this command, the turtle's pen was lifted off the screen; the effect was to eliminate the turtle tracks as it moved along. Our formal definition of PENUP is as follows:

PENUP	**(takes no inputs)**
Purpose	Takes turtle "up" off screen; tracks will not be displayed
Abbrev	PU
Inputs	None
Outputs	None

Since PENUP takes no inputs and gives no outputs, it is the simplest type of Logo primitive. Others that operate the same way are SHOWTURTLE, CLEARSCREEN, WINDOW, WRAP, and FENCE.

In this chapter, we used the primitive ROUND as an example of an operation. It is different from commands like FORWARD and PRINT in that it outputs a value. Here is the formal definition of ROUND:

ROUND	**(takes 1 input)**
Purpose	Rounds off decimal input to nearest integer
Abbrev	None
Inputs	Any real number
Outputs	Real integer

The operation ROUND would take an input like "4.231" and output "4."

Our final example of formal definitions for primitives we've used is REPEAT. Like FORWARD, it is a command, but unlike other commands we've used this far, it takes two inputs. Here is the formal definition of REPEAT:

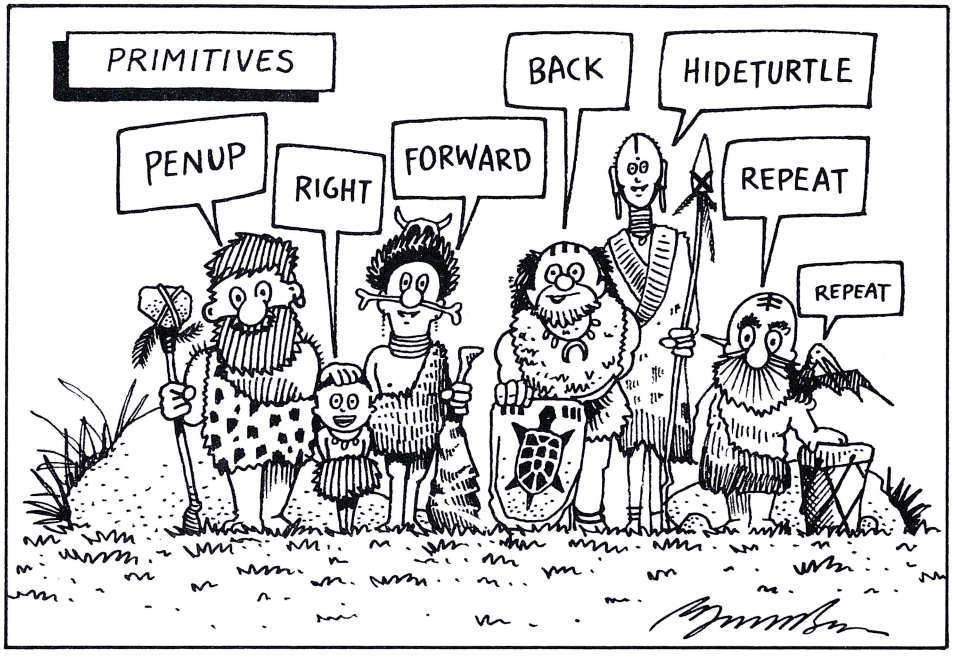
REPEAT	**(takes 2 inputs)**
Purpose	Repeats a specified number of times the command that follows
Abbrev	None
Inputs	First input is a number; second input is a list
Outputs	None

Examples of procedure lines containing REPEAT would be as follows:

```
REPEAT 4 [FORWARD 30 RIGHT 60]
REPEAT 7 [PRINT [SEVEN IS MY LUCKY NUMBER]]
```

The two inputs for each REPEAT command are a number followed by a list. Notice that lists are set off in brackets.

For the next few chapters, we include formal definitions of most of the new Logo primitives introduced. We do this to emphasize inputs and outputs and to remind you of Logo's basic structure.



3.4 Summary

Here are some of the important points in this chapter.

1. Primitives or procedures and their inputs form the structural basis of Logo.

2. There are two types of primitives: commands that do something, and operations that do something and give outputs that are then fed to commands or operations.
3. There are five types of inputs: numbers, words, lists, dotted words, and operations.
4. Dotted words provide a way to define procedures with variable inputs. They are also used to pass values on to primitives like FORWARD.
5. Operations are another form of input. Operations are primitives, like ROUND, or procedures that we write that produce an output.
6. Formal definitions of Logo primitives, which include information on purpose, inputs, and outputs, help us to develop a more clear understanding of how to "communicate" in the language.

3.5 Review Questions and Exercises

Review Questions

1. Every line in a Logo program starts with what?
2. How many inputs does REPEAT take?
3. Give an example of a primitive that takes zero inputs.
4. What is the difference between an operation and a command?
5. Give an example of an operation.
6. What does Logo do if you type ROUND 5.666?
7. How many kinds of primitives are there?
8. How many kinds of inputs are there?
9. Why are dotted words so useful as inputs?
10. What kind of unquoted words can be thought of as inputs? What kind cannot be used as inputs?

Exercises at the Terminal

1. Try to make Logo give you the following error messages.

```
NOT ENOUGH INPUTS TO REPEAT
I DON'T KNOW WHAT TO DO WITH 35
I DON'T KNOW HOW TO WOOF
```

2. Try out the procedure RECTANGLE :HEIGHT :LENGTH described in the chapter.
3. Create a procedure called MODEL.H that uses RECTANGLE and TRIANGLE and that has two inputs to draw houses (rectangles with triangle roofs on them) of different heights and lengths.
4. Experiment with the MODEL.H procedure to draw a row of houses of different sizes that look like a city block. Can you put this in a procedure called CITY?
5. We've drawn closed shapes using REPEAT as follows:

```
REPEAT 3 [FORWARD :SIZE RT 120]
REPEAT 4 [FD :SIZE RT 90]
```

   Can you draw a five-sided figure? How about six sides?
6. Can you create a procedure called POLYGON that takes two inputs, :SIZE and :NUMBER and draws a closed symmetrical shape with any "number" of sides? (Hint: you will need to use the variable :NUMBER twice.)

Answers to Review Questions

(1) Every line in a Logo program starts with a primitive (command) or a procedure. (2) REPEAT takes two inputs; the first is a number, the second is a list. (3) Primitives which do not take inputs include SHOWTURTLE, CLEARSCREEN, FENCE, WINDOW, WRAP, PENDOWN, PENUP, FULLSCREEN, SPLITSCREEN, and TEXTSCREEN. (4) Primitives that produce an output are called "operations"; those that do not are called "commands." (5) Operations include ROUND, SUM, and PRODUCT. (6) Logo would respond with I DON'T KNOW WHAT TO DO WITH 6. (7) There are two kinds of Logo primitives—commands and operations. (8) There are five kinds of inputs: numbers, words, lists, dotted words, and operations. (9) Dotted words as inputs allow us to write variable procedures. That is, we can write generic procedures for drawing pictures of squares of different :SIZE. When we run the procedure, we supply a number input. (10) Operations or procedures that produce an output are unquoted words that can be thought of as inputs. Commands are unquoted words that can not be used as inputs.

CHAPTER 4

CREATING AND MANAGING YOUR RESOURCES

CONCEPTS:

initializing disks
creating utilities
Logo's workspace
saving your work
handling disks
using a printer

PRIMITIVES:

POTS, PO, POALL, ERASE, ERALL, PACKAGE, BURY, SAVE, LOAD, ERASEFILE, CATALOG, .PRINTER

4.0 Introduction

In the book *The Third Wave,* Alvin Toffler coined the term "cottage industry" to characterize the changing American workplace. Toffler visualizes the time when large numbers of people will no longer commute to their offices five days a week in order to handle information chores. Instead, they will be able to use computers to create a "workspace" at home. Information will be brought electronically from the office into the cottage workspace, processed, then sent back to the office.

Logo provides us with a similar, small-scale version of the cottage

workplace. When Logo is first booted, we are placed at the Command Level. In addition, Logo creates a workspace within the Apple for temporarily storing our procedures and variables. When we have finished processing information, that is, creating and editing procedures, we can send our workspace contents to permanent storage onto a "floppy disk." Similarly, Logo provides primitives for transferring disk information back to our workspace.

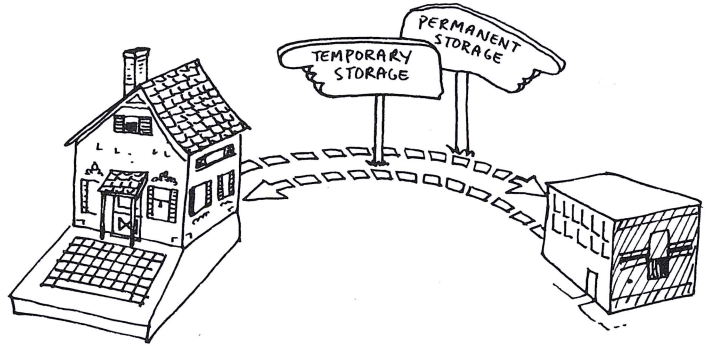


Fig. 4-1. Logo as a cottage workspace.

Once we have permanent storage facilities on disk, one of the first kind of procedures we'd like to save are called graphics "utilities." Utilities are procedures that can be used over and over again in a variety of graphics designs. They become important "tools" for building graphics programs. We have already created two such resources, RECTANGLE and SQUARE. In this chapter we'll create a few more utilities and then learn how to package them for regular transfer between our workspace and disk.

We will begin this chapter by "initializing" a disk so that the disk is compatible with the Apple computer. This is required before we can save information onto the disk.

4.1 Initializing a Disk to Save Work

In this section, we'll see how to prepare a disk for saving procedures. The words "disk," "diskette," and "floppy disk" all refer to the same thing: a square, thin, black, 5 1/4-inch plastic storage device. We'll show you how to initialize or "format" a disk on which to save your Logo files and introduce the commands that control the Apple Disk Operating System, called "Apple DOS."

You may remember that when we first booted Logo into Apple, we received the following message on the screen:

```
PRESS THE RETURN KEY TO BEGIN


IF YOU HAVE YOUR OWN FILE DISKETTE,
INSERT IT NOW. THEN PRESS RETURN




(C) LOGO COMPUTING SYSTEMS, INC. 1982
```

Not just any old diskette will do. Before you can insert a disk, it must be formatted or initialized to be compatible with Apple DOS.

4.1.1 Initializing a Disk

Caution: when you initialize a disk, everything on the disk is erased. Do NOT initialize a disk if you think it may contain useful information.

To initialize a disk, we need to exit Logo. Turn the Apple off by pressing the rocker switch at the left rear corner. The power light will go off. Remove the Logo diskette from the drive and put it back into its protective envelope. Take the Apple DOS 3.3 System Master diskette which came with the Apple and insert it into the drive. Restart the Apple by depressing the rocker switch. The familiar whirr will begin, followed by this display:

```
DOS VERSION 3.3       A Date


System Configuration SYSTEM MASTER

]□
```

Applesoft BASIC prompt

The flashing cursor next to a right bracket sign is your cue that the system will accept an initialization process. Remove the System Master diskette and replace it in the drive with a blank diskette. To initialize the blank diskette, type the following:

```
]INIT HELLO
```

and press RETURN. The red light on the disk drive will go on, and there will be a series of whirrs and clacks for several seconds. When the red light goes off, your disk will be formatted for accepting Logo procedures as "files." With a felt tip pen, write a descriptive name (like Logo File Diskette #1) along with your name and date on the disk label.

Now remove the newly initialized diskette, turn off the Apple, insert the Logo diskette and reboot Logo. When the message concerning your file diskette appears, remove the Logo diskette and insert yours. Press RETURN and your electronic cottage is ready to go.

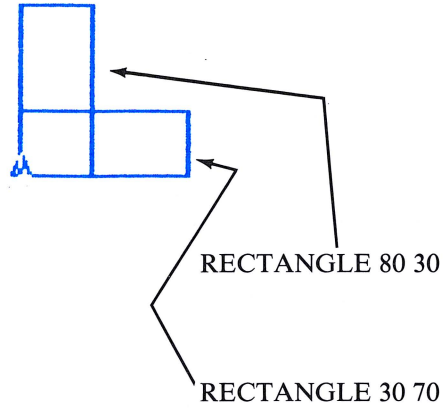
4.2 Graphics Utilities

Utilities are like tools in a workshop. They are procedures that are used over and over again to help build complex programs. Those that can be incorporated in our turtle designs are called "graphics utilities." In this section, we'll create several procedures that will be useful utilities. Also, by typing the procedures in, we'll be able to demonstrate "workspace management." Later in this chapter, these designs will be saved to the file diskette created above.

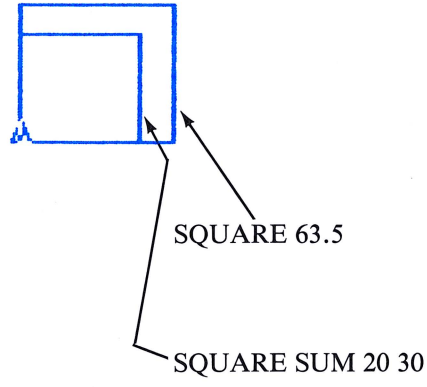
Our first three graphics utilities are procedures written earlier in this text. They are utilities for drawing rectangles, square, and triangles, as follows:

```
>TO RECTANGLE :HEIGHT :LENGTH
>FD :HEIGHT RT 90
>FD :LENGTH RT 90
>FD :HEIGHT RT 90
>FD :LENGTH RT 90
>END
```

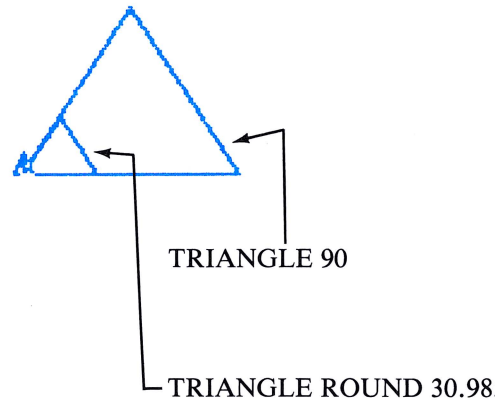
```
?RECTANGLE 80 30
?RECTANGLE 30 70
```




```
?TO SQUARE :SIZE
>REPEAT 4 [FD :SIZE RT 90]
>END

?SQUARE 63.5
?SQUARE SUM 20 30
```



```
?TO TRIANGLE :SIDE
>RT 30
>REPEAT 3 [FD :SIDE RT 120]

>LT 30
>END

?TRIANGLE 90
?TRIANGLE ROUND 30.985
```

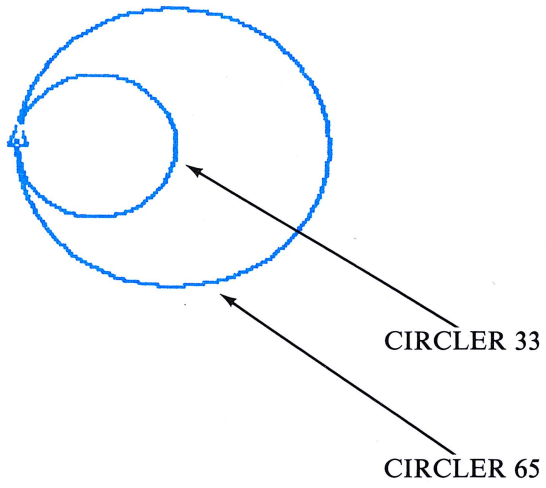


The next three graphics utilities involve circles and arcs. Apple Logo comes with utilities called "AIDS" which produce curved figures. We prefer to develop our own. You may remember that the circumference of a circle is equal to two times Pi times the radius. We'll first write a procedure which will produce the numerical value for Pi each time we want to draw a circle:

```
?TO PI
>OUTPUT 3.14159
>END
```

We will explain the primitive "OUTPUT" in Chapter 8. For now, you can consider that it causes the procedure PI to act as an operation as discussed in Chapter 3. In order to draw a circle, the turtle must travel a distance equal to the circumference of a circle, 2 * PI * :RADIUS, while traveling a total of 360 degrees. We'll call this procedure "CIRCLER" or "CIRCLE to the Right." The listing shows how to accomplish this:

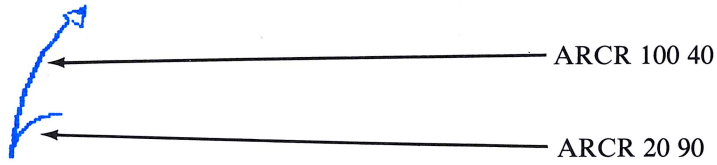
```
?TO CIRCLER :RADIUS
>REPEAT 36 [RT 5 FD 2 * PI * :RADIUS /36 RT 5]
>END
 ?CIRCLER 33
 ?CIRCLER 65
```



Notice that we turn right 5 degrees and then only go forward a distance 2*PI*RADIUS/36, then turn 5 degrees and repeat this process 36 times. The reason for the two 5-degree turns is to make the circle more symmetrical.

The last utility, ARCR, is similar to the CIRCLER program. ARCR ("ARC Right") draws arcs of varying sizes starting to the right of the turtle's location. You can draw arcs of 20, 30, 40, degrees, etc., but not 45 degrees. This is because REPEAT truncates its first input to an integer.

```
?TO ARCR :RADIUS :DEGREES
>REPEAT :DEGREES/ 10 [RT 5 FD 2 * PI * :RADIUS/ 36 RT 5]
>END

?ARCR 20 90
?PU
?HOME
?PD
?ARCR 100 40
```



As an exercise, write procedures CIRCLEL and ARCL which will draw circles and arcs to the left.

Now that we have some useful procedures, let's find out how to manage our workspace and save these utilities for further use.

4.3 Logo's Workspace

In chapter 1, we learned how to boot Logo into Apple's memory. A considerable amount of Apple's memory (about 3/4th of the 64K) is taken up by Logo; what remains is called the "workspace." When we create procedures and establish variables, this information is housed in the workspace.

When the Apple is turned off, everything in its memory is lost. At first, this might sound like a real problem—sort of like writing with disappearing ink! However, we can permanently store all the information in Apple's workspace onto a disk. The ink may disappear, but the carbon copy will still be readable. Before we learn to save to disk, though, we'll investigate some Logo primitives for workspace management.

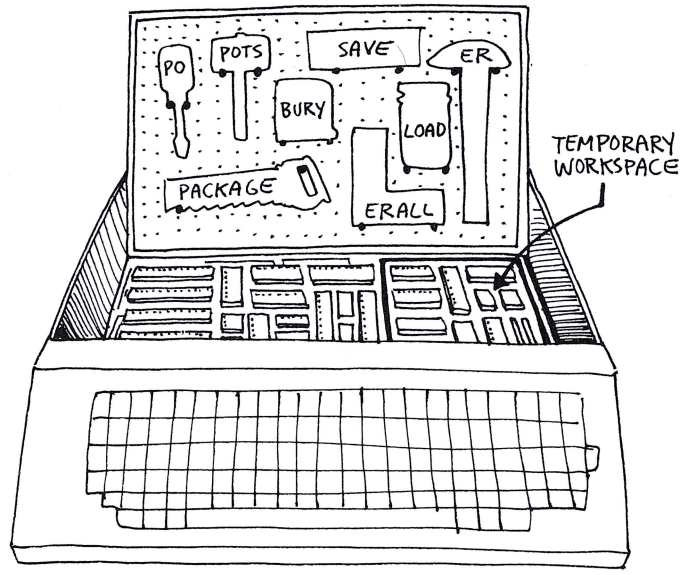


Fig. 4-2. The workspace.

4.3.1 Managing Procedures in the Workspace

Once Logo has procedures in the workspace, we can use the primitive POTS to list the workspace contents. POTS stands for "Print Out TitleS." For example, if our utilities are in the workspace and we type

```
?POTS
```

the following will be displayed:

```
TO ARCL :RADIUS :DEGREES
TO CIRCLEL :RADIUS
TO ARCR :RADIUS :DEGREES
TO CIRCLER :RADIUS
TO PI
TO TRIANGLE :SIDE
TO SQUARE :SIZE
TO RECTANGLE :HEIGHT :LENGTH
```

Here is a formal definition of POTS:

POTS	**(takes no inputs)**
Purpose	Directs Logo to display the titles of all procedures currently in the workspace
Abbrev	None
Inputs	None
Outputs	None

Another workspace management primitive is PO. PO stands for "Print Out"; it directs Logo to display the program code for each procedure you name. For example, to see the listing for SQUARE, direct Logo to

```
?PO "SQUARE
```

The procedural description for SQUARE will be displayed on your screen. Later in this chapter, we'll show you how to use PO to obtain a paper or "hard copy" of a procedure if you have a printer. Here is the formal definition of PO:

PO	**(takes one input)**
Purpose	Directs Logo to list procedure definition

Abbrev	None
Inputs	One; either the name of a procedure, which must be quoted, or a list of procedure names
Outputs	None

Here are two examples you can try with the information currently in the workspace:

```
?PO "SQUARE
?PO [SQUARE RECTANGLE PI]
```

A variation of PO is POALL. POALL stands for "Print Out ALL." When we direct Logo to

```
?POALL
```

the lines of code for all procedures in memory will be displayed, as well as any variables created. Like POTS, POALL takes no inputs.

4.3.2 Erasing Procedures in the Workspace

We don't have to wait until the Apple is turned off to remove information from the workspace. Logo has a primitive with an eraser attached; it will remove whatever procedure you want without affecting other procedures. It's called, quite appropriately, ERASE. Like PO, it takes a procedural name as an input. It would be used like this:

```
?ERASE "TRIANGLE
```

Either POALL or POTS (POTS is simpler) can be used to verify that the TRIANGLE procedure has indeed been erased. Here is the formal definition of ERASE:

ERASE	**(takes one input)**
Purpose	Erases or eliminates a procedure from workspace
Abbrev	ER
Inputs	One, either the name of a procedure, which must be quoted, or a list of procedure names
Outputs	None

To remove all procedures in the workspace, it is possible to "erase 'em all." Logo's primitive for this is ERALL.

ERALL needs to be used with caution, for obvious reasons.

4.3.3 The Diskette: Permanent Storage

It is quite easy to copy all procedures currently in your workspace onto the storage diskette using the Logo primitive SAVE. First, let's make sure our graphics utilities are currently in Apple's memory by using the command POTS. Retype any procedure that you may have erased. To save these utilities to disk, type

```
?SAVE "GRAPHICS
```

and press RETURN. The familiar red light and whirrs will be your audio indication that GRAPHICS are being stored on the disk; the display

```
8 PROCEDURES SAVED
?
```

when it's finished, will be your visual confirmation. All eight utilities were saved (we assume you added CIRCLEL and ARCL). In fact, all information currently in the workspace would have been saved under the name GRAPHICS. For that reason, it's sometimes a good idea to PACKAGE selected procedures, and SAVE that package as described in the next section. If you don't package, you might save extraneous and useless information that has been accumulating in Apple's memory. Another way around the problem would be to ERASE all the information you don't want from the workspace, and then SAVE the rest.

The primitive CATALOG (no inputs) can be used to confirm that SAVE worked as it should. CATALOG directs DOS to display the current titles of all files on the disk. Type

```
?CATALOG
```

The drive will whirr, and the following will be displayed:

```
DISK VOLUME 254


A    6 HELLO
T    3 GRAPHICS.LOGO
```

The letters A and T stand for "Applesoft" and "Text." The numbers which follow are an indication of the program size. The word ".Logo" following the program name indicates that the information in that text file is written in Logo.

To load the contents of your diskette into Logo's workspace, use the command LOAD followed by quotation marks (") and the name of the file you wish to load. Assuming you've correctly saved the GRAPHICS, let's shut the system down, reboot Apple, and then place your file diskette into the drive when asked. Now, direct Logo to

```
?LOAD "GRAPHICS
```

Don't add the ".Logo" part of the file name. Logo does that automatically. Whirrs will accompany the process of transferring the information from the disk into the Apple. When the red disk drive light goes off, check your workspace with the command, POTS:

```
?POTS
TO ARCL :RADIUS :DEGREES
TO CIRCLEL :RADIUS
TO ARCR :RADIUS :DEGREES
TO CIRCLER :RADIUS
TO PI
TO TRIANGLE :SIDES
TO SQUARE :SIZE
TO RECTANGLE :HEIGHT :LENGTH
```

Aha! The Logo cottage is now functioning. We can transfer information from the disk to our workspace and back again.

One final command will give us better control of our workspace disk file system. If we want to delete a file from the disk, use the primitive ERASEFILE followed by quote marks (") and the name of the file you wish to erase. For example, if you now typed

```
?ERASEFILE "FILENAME
```

and press RETURN, you will eliminate the Logo file FILENAME from your disk. DON'T DO THIS UNLESS YOU REALLY MEAN IT! If you did erase a file, you could confirm its elimination by directing Logo to CATALOG the contents of your disk.

As you write and save larger procedures and programs, you'll want to make "back up" disks to protect your creative investment. You can do this by saving workspace contents twice onto two different initialized disks. (You can also make copies of your Logo file diskettes. Refer to your Apple manual for directions on making copies of disks.)

4.3.4 File Names

Before we leave this section, we'd like to suggest a few hints on file names. In our example we saved eight procedures under the file name "GRAPHICS." We could have named our file almost anything that we wanted, with a few limitations. We recommend that you call the file by a name that will be meaningful to you. Also, Apple Logo requires that file names be limited to 30 characters. If you give your file a name over 30 characters, Logo will only "read" the first 30 characters. For example, if you direct Logo to

```
?SAVE "DESIGN.BY.JOHNNY.B.MASTER.PROGRAMMER.ON.SEPT.18.1984
```

the file name will consist of only the first 30 characters:

```
DESIGN.BY.JOHNNY.B.MASTER.PROG
```

In addition, Logo will not accept file names that already exist on the disk. If, for example, you LOAD and modify GRAPHICS, and then direct Logo to SAVE "GRAPHICS it will respond with

```
FILE ALREADY EXISTS
```

To get around this problem, you can

```
?ERASEFILE "GRAPHICS
```

and then

```
?SAVE "GRAPHICS
```

A second way around this would be to call your second version of the program by a different name, say GRAPHICS.2 or GRAPH.TWO.

Finally, as with procedure names, file names need to be a continuous string of characters. Spaces are not acceptable in file names. If you directed Logo to

```
?SAVE "MY FAVORITE PROGRAM
```

it would respond with

```
I DON'T KNOW HOW TO FAVORITE
```

and wouldn't save anything to disk. Here is a summary of suggestions for creating file names:

Naming files

1. Choose a descriptive name.
2. Don't use names over thirty characters in length.
3. Do not leave blank spaces between characters.
4. Erase unwanted procedures in the workspace, or package your procedures to prevent saving unnecessary information to the disk.
5. Give new names for new files or erase old files *before* saying the workspace under the same name.

4.4 Handling Your Disks

The information stored on your disk represents your library of utilities and procedures. Like a collection of good books, tapes, and albums, this storage medium requires considerable care if it is to remain useful to you. Books shouldn't be left out in the rain; tapes and albums fare badly in extreme heat. Your diskettes are even more sensitive. However, if you treat each one as a prized book or classic record album, the disk and information on it will last for a long time.

First a word about holding the disks. The best technique is to handle them on the outer edge only without squeezing them. You should place the disk on the disk drive as shown in Fig. 4-3.

Your thumb should be on the upper left hand side of the disk, over the label. Be sure not to touch the exposed disk surface near the donut hole in the center. We recommend not smoking or eating while handling disks. You should never write on a diskette with a pencil or ball point pen; always use a felt tip pen. Also, don't write on paper on top of a diskette.

There's an old adage that the only two places a diskette should be found are in its protective paper sleeve inside a box standing vertically, or inside the disk drive. Removing a diskette from the drive and casually leaving it about invites disaster.

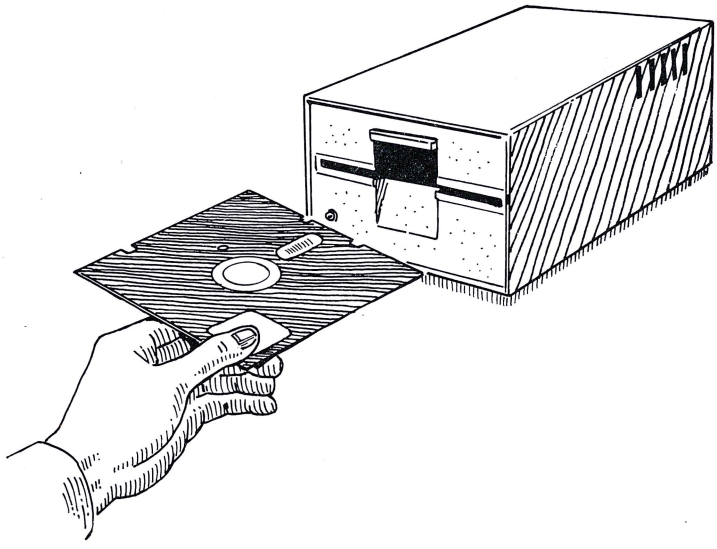


Fig. 4-3. Inserting disk into drive.

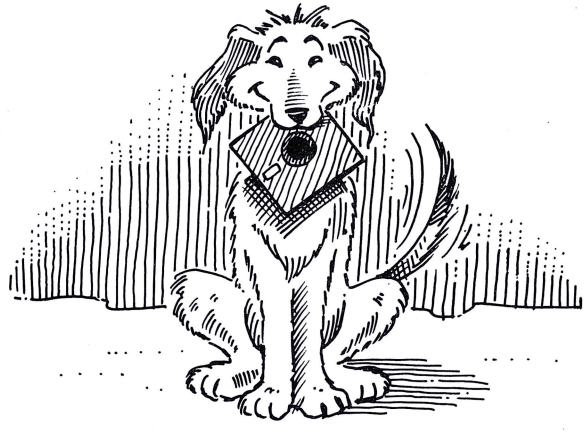


Fig. 4-4. Smiling dog with Doggy Diskette.

The boxes themselves should be placed in a protected environment. Safe places do NOT include radiators, open window ledges, or eyes of hurricanes! Even with the best of care, however, things sometimes do go wrong. We recommend you make backup copies of your favorite files and disks.

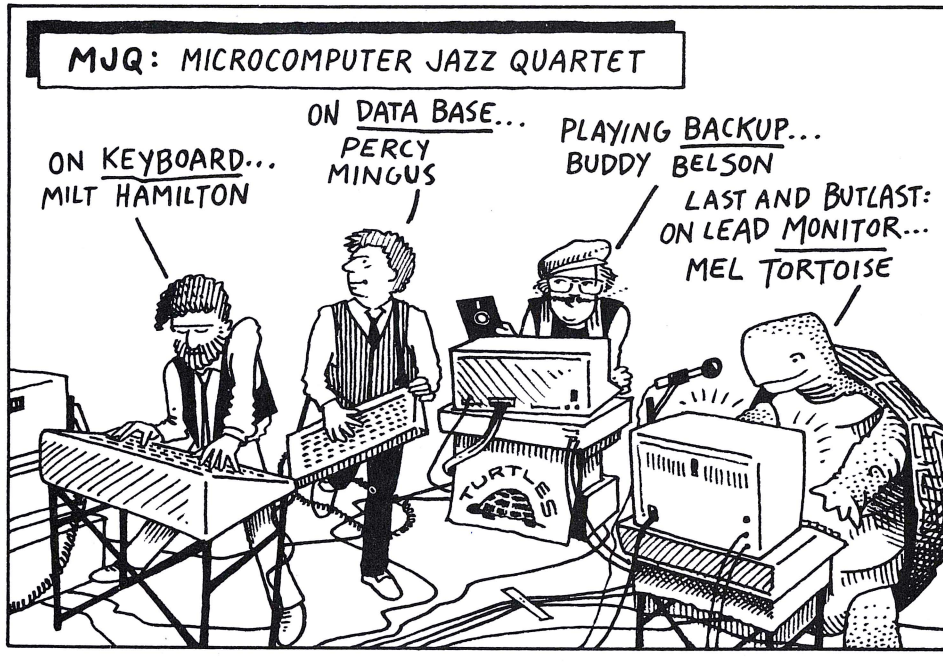
Finally, diskettes store information using tiny magnets coated on the disk. Therefore, they should be kept away from permanent magnets and sources of magnetic fields like television sets and motors.

Do not try to clean diskettes. If you suspect dirt on a diskette, try to make a backup copy on a fresh diskette and throw the dirty one away.

In summary, disk management hints include the following:

Protecting Your Disks

1. Hold diskette only by the edges.
2. Store disks in their protective sleeves vertically inside a box.
3. Use a felt tip pen to label diskettes.
4. Do not expose the disks to extreme temperatures, smoke, dust, or permanent magnets.
5. Make backup copies of all disks.


MJQ: MICROCOMPUTER JAZZ QUARTET
ON KEYBOARD... MILT HAMILTON
ON DATA BASE... PERCY MINGUS
PLAYING BACKUP... BUDDY BELSON
LAST AND BUTLAST: ON LEAD MONITOR... MEL TORTOISE
TURTLES

4.5 Packaging Procedures and Burying Them

The material in this section is most useful for large programs. Beginning users can skip this for now.

There may be times when we want Logo to gather several procedures together and keep them out of sight (but still available). If these procedures are "packaged" together and then "buried," they'll be hidden from workspace primitives like POALL, ERALL, and SAVE. To do this, we use the Logo primitives PACKAGE and BURY. Here is the formal definition of PACKAGE.

PACKAGE	**(takes two inputs)**
Purpose	Puts specified procedures into single package
Abbrev	None
Inputs	Two; the first is the package name, the second is a procedure or a list of procedures
Outputs	None

Here are two examples of PACKAGE with utilities which are currently in our Apple workspace:

```
?PACKAGE "BOX "SQUARE
?PACKAGE "BOX [SQUARE RECTANGLE]
```

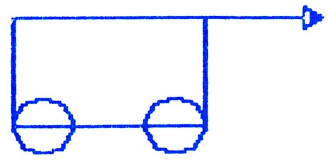
In the first example, the procedure SQUARE would be packaged into BOX; in the second example both SQUARE and RECTANGLE would be placed into BOX.

The second primitive is BURY. Here is its definition.

BURY	**(takes one input)**
Purpose	"Hides" all procedures in the package from workspace primitives ERALL, POTS, POALL, and SAVE
Abbrev	None
Inputs	One; the package name as a quoted word
Outputs	None

You could, for example, BURY the packaged procedures you had created above in BOX. Why do we PACKAGE and BURY? The main reasons are as follows. First, it allows us to save some information in the workspace without saving all procedures in it. Secondly, it allows us to save and load groups of related procedures. Another reason is to prevent the altering of files by another computer user.

As an example, suppose we wanted to construct a wagon as shown below.



First package all of our utilities into a single file:

```
?PACKAGE "GRAPH "GRAPHICS
```

Now bury the package:

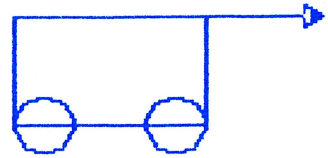
```
?BURY "GRAPH
```

If you now direct Logo to ERALL and POTS, you'll find there is no listing of procedures in memory. Actually, all of our GRAPHICS procedures are still there, but they're hidden from ERALL and POTS. Now write the program for WAGON:

```
?TO WAGON :HEIGHT :LENGTH
>RECTANGLE :HEIGHT :LENGTH
>CIRCLER :HEIGHT/4
>PU
>RT 90
>FD :LENGTH
>PD
>LT 90
>CIRCLEL :HEIGHT/4
>PU
>FD :HEIGHT
>PD
>RT 90
>FD :LENGTH/2
>END
```

To draw WAGON, provide inputs for :HEIGHT and :LENGTH when you call the procedure:

```
?WAGON 50 80
```



Now SAVE "WAGON. After the program has been put onto the disk, Logo will inform you that one procedure has been saved; BURY also hides procedures from SAVE! If we had not first packaged and buried the GRAPHICS, all of the information we had in Apple's memory would have been saved to disk. Since the GRAPHICS were not saved with WAGON, when you want to run that program in the future you'll need not only to LOAD "WAGON but also to LOAD "GRAPHICS. (We could have saved all the utilities along with WAGON to avoid this last step. In this simple example there's not much difference either way.)

Here is a summary of all the Logo workspace primitives used thus far.

Table 4-1. Logo Workspace Primitives

PRIMITIVE	FUNCTION
`POTS`	displays the title of all procedures in memory
`PO` name or `PO` [name1 name2]	displays program listing of procedure or procedures
`POALL`	displays the titles and program listings for all procedures in memory and all variables values
`ERASE` name or `ERASE` [name1 name2]	erases procedure(s) named from memory
`PACKAGE` name1 name2 or `PACKAGE` name1 [ ]	combines procedures named into a single package called name1
`BURY` name	hides package named from operating system primitives POTS, POALL, ERALL, and SAVE

4.6 Obtaining a Hard Copy

In addition to memory and disk management, Logo allows us to make paper or "hard" copies of our procedures and files. Depending on your printer's capabilities, you might be able to make hard copies of the actual drawings displayed on your screen. You can skip this section if you don't have a printer.

It is fairly easy to obtain a paper or hard copy of procedure definitions, as we'll see next. The Logo command is .PRINTER followed by the slot number that your printer interface card is located in. For example, if your printer interface card is in slot #1, the following command will activate your printer (assuming that it is plugged in and turned on!).

```
?.PRINTER 1
```

Now all communications between you and Logo appears on the printed page rather than on the video screen. You can now direct Logo to print out listings with either "print out" command, PO or POALL. You can also list your files with the command CATALOG.

Note that the familiar prompt ("?") disappears when the printer is activated with .PRINTER 1, though the flashing cursor remains. When you've finished getting a hard copy of everything you want, deactivate the printer with the command

```
?.PRINTER 0
```

Notice that the prompt is back "on line" when the printer goes off.

If you want to obtain a paper printout of a turtle graphics design you need to have a printer capable of printing graphics as well as a graphics interface card. We'll focus our comments on two printer types for printing turtle graphics: an Epson MX 80 with Graffax and a Prowriter 8510. We'll also assume you have the Grappler+ printer interface card. If you have different hardware, refer to the equipment user's manual for directions. Finally, we'll assume that the DIP ("Dual Inline Pin") switches have been properly set according to the manufacturer's instructions.

Now boot Logo, and LOAD "WAGON and "GRAPHICS from your file diskette. In order for us to obtain a hardcopy of the graphics produced by WAGON, we need to write the following procedure:

```
?TO HARDCOPY
>.PRINTER 1 <———————— Notice that commands to activate
```

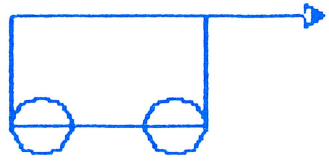
```
>(TYPE CHAR 9 "G CHAR 13)
>.PRINTER 0
>END
```

and deactivate the printer are included

Save HARDCOPY to disk so that you can use it whenever you want to print out your graphics design. Now, with WAGON, GRAPHICS and HARDCOPY all in the workspace, direct Logo to put a design on the screen with the following:

```
?WAGON 40 90
```

The following will be displayed on your screen.



Make sure the printer is on and type in the following:

```
?HARDCOPY
```

Your printer will draw the design on the paper.

Let's try an experiment. Clear the screen with CLEARSCREEN and again direct Logo to

```
?HARDCOPY
```

Oops! You'll obtain nothing on your paper except the small triangle in the center of the screen; the turtle is resting at home. You need to have a graphic display on the screen before you direct Logo to HARDCOPY it.

If you refer to your Grappler+ manual, you'll see you can write modified versions of HARDCOPY which will allow you to change the size of the drawings, or to rotate them, or even to produce "negatives."

4.7 Summary

Here is a summary of the important points in this chapter.

1. The Logo language, and any procedures and variables you type in, are housed in the "workspace."

2. To save procedures, you need to have an initialized disk. When you save procedures, all information in the workspace that is not "buried" will be saved onto the disk.

3. It's convenient to group frequently used procedures into packages and save them to disk. When you need the tools, you can load those procedures from the disk into the workspace.

4. Logo workspace primitives include

PRIMITIVE	EXAMPLE
POTS	
ERALL	
POALL	
PO	PO "WAGON or PO [WAGON PI ARCR]
ERASE	ERASE "WAGON or ERASE [WAGON PI ARCR]
PACKAGE	PACKAGE "BOX "SQUARE or PACKAGE "BOX [SQUARE LONG]
BURY	BURY "BOX

5. Disk management primitives include

PRIMITIVE	EXAMPLE
SAVE	SAVE "GRAPHICS
LOAD	LOAD "GRAPHICS
CATALOG	
ERASEFILE	ERASEFILE "WAGON

6. To activate the printer, type in .PRINTER 1 and press RETURN. To stop the printer, use .PRINTER 0 and RETURN.

4.8 Review Questions and Exercises

Review Questions

1. What do we mean by "initializing" a disk? When you do an initializing routine, what happens to any information that may be on the disk?
2. What do we mean by the term "workspace"?

3. What do the following stand for?
   a. PO
   b. POTS
   c. POALL
   d. ERALL
4. What happens to all (unburied) procedures in memory if you ERALL?
5. If ARCR calls PI and you ERASE "ARCR, do you also erase PI?
6. Which of the following do not take any inputs?
   a. ERALL
   b. ERASE
   c. ERASEFILE
   d. POTS
   e. CATALOG
   f. SAVE
   g. BURY
7. There are two types of errors in the following line. What are they?

   ?SAVE I DID THIS DRAWING

8. Why is it a good idea to make back up copies of your disks?
9. List the sequence you'd use to obtain a hardcopy picture of a procedure called FLOWER :SIZE.
10. What is the difference between ERASE name and ERASEFILE name?

Exercises at the Terminal

1. There are two ways to store utilities
   a) store them all together under one name, such as GRAPHICS as we described in this chapter.
   b) store them individually.

   We recommend *both* ways.

   As an exercise, save each graphics utility onto disk under its own name.
2. Write a utility procedure to create hexagons of varying sizes. Include that procedure in your GRAPHICS utilities.
3. Use CIRCLER, CIRCLEL, ARCR, and ARCL to draw flowers with petals or "happy face" drawings. Save to disk only the information needed to create these figures.

Answers to Review Questions

(1) When you initialize a disk, you prepare it to accept Logo commands for saving and loading procedures. A new disk needs only to be initialized once; if you initialize it again, you erase everything (!) on the disk. (2) The workspace is the amount of memory available in the Apple computer after the Logo language has been loaded. (3) a. Print Out b. Print Out TitleS c. Print Out ALL d. ERRase All (4) They are all erased. (5) No, PI would still be in the workspace, assuming you had written it as a separate procedure. (6) ERALL, POTS, CATALOG (7) Procedure names must be a continuous string, and SAVE takes a quoted word. Thus, the correct line would be SAVE "I.DID.THIS.DRAWING. (8) Disks sometimes fail. If you have only one copy, you risk losing all of your work. (9) LOAD "HARDCOPY, turn on printer, call FLOWER 40, call HARDCOPY. (10) ERASE name removes a procedure or program in the workspace; ERASEFILE name deletes it from the disk.

CHAPTER 5

DEVELOPING GOOD PROGRAMMING TECHNIQUES

CONCEPTS:

stages of programmer development
top-down programming design
the box chart
stepwise refinement

Alice and the Cheshire Cat
"Would you tell me, please, which way I ought to go from here?"
"That depends a good deal on where you want to get to," said the Cat.
"I don't much care where—" said Alice.
"Then it doesn't matter which way you go," said the Cat.
(Lewis Carroll, Alice in Wonderland)

5.0 Introduction

Imagine that you've recently moved to a new city. On your first free day, you decide to explore your new environs. You drive along new streets and enjoy discovering what each block has to offer. You begin to classify regions of the city in terms of distinctive activities: the west bank of the river seems to be the financial district; "restaurant row" is found on the main north/south boulevard, etc. The next time you need to accomplish some business downtown, you feel much more confident in your ability to find your way around. You still will make wrong turns, and will frequently ask for directions from a friendly "native," but you're feeling more at home.

Suppose that after several weeks of such experiences, you face a day filled with tasks that require considerable organization on your part. You're

scheduled for a hair cut at 10:00, a conference with a stockbroker at 1:00, fitting for ski boots at 2:30, you need to pick up clothes from the cleaners before they close—the list goes on and on. You might decide this is the day that preplanning, with paper, pencil, and map is essential. You set to work, and in a short time based on your knowledge of the city, you effectively map out your route for the next day. A month ago, such an undertaking would have been impossible.

In Logo, we go through similar stages, from exploration and discovery to systematic planning, as we develop our programming abilities.

5.1 Three Stages of Programming Skill in Logo

The process of learning to program might be viewed as going through three stages of development. In Logo these stages can be characterized by:

Stage 1–exploration and discovery
Stage 2–use of variables in simple procedures
Stage 3–top-down design of programs.

In Stage 1, the beginning computer user has a wonderful time exploring Logo's turtle graphics, learning many graphics primitives, and creating magnificent drawings. Logo provides opportunities in Stage 1 unmatched by any other language.

In Stage 2, the programmer is learning the full Logo vocabulary and how to write procedures containing five to fifteen lines, how to write procedures with variables and inputs, and how Logo's structure fits together and is used to write super procedures.

Stage 3 involves top-down program design. Now, the maturing programmer has to plan the program with paper and pencil. Sometimes, it is not easy to make the transition to this stage. The techniques that create so much initial interest in Logo—sitting at the terminal, playing, trying out ideas and the like—are no longer adequate for writing complex programs with five to fifteen "boxes" or subprocedures. The excitement of programming at this level is not the same as with the initial discoveries of turtle graphics.

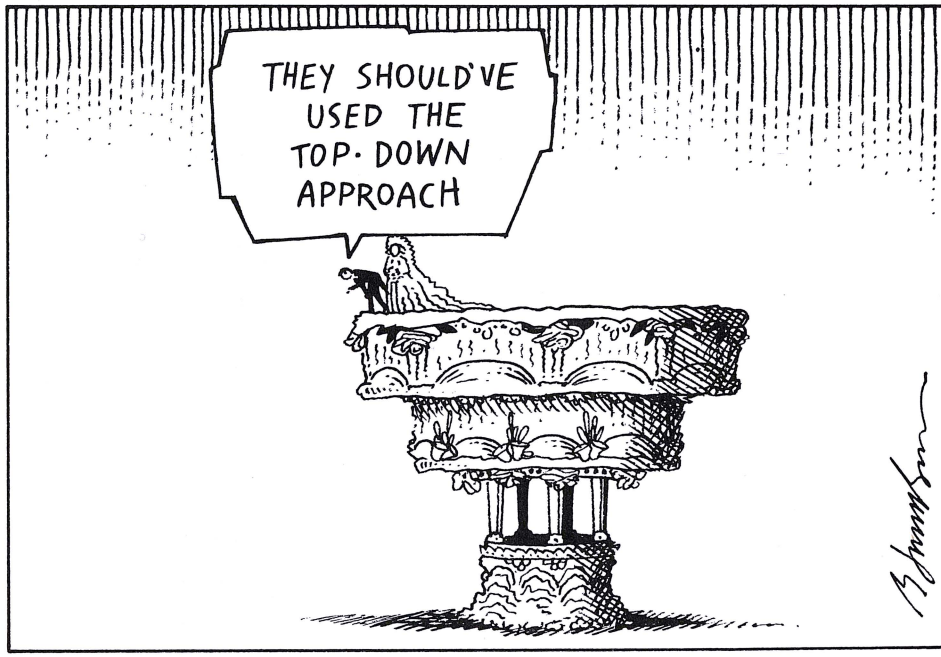
But, once you know Logo's language and structure, once you begin to plan and create complex programs that do what you want, there comes a deep feeling of the "joy" of understanding. You can write programs to do whatever you want. You have opened a door to another world. The excitement of Logo comes back again in a new form. We have experienced it in

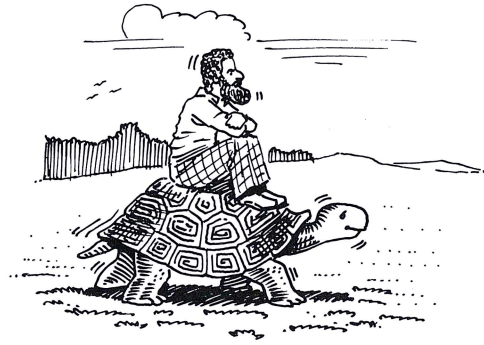
writing this book and we hope to share it with you as you learn this marvelous language.

5.2 Programming and Problem Solving

One characteristic feature of the human condition is that we have a diversity of intellectual approaches to solving problems. Unlike most forms of life on Earth, we are not instinctively locked into "either/or" options. When confronted with the most trivial to the most profound problems (from "Shall I comb my hair?" to "What is the best way to educate children?") we have a variety of methods available to us. Our language is rich in words describing approaches: we can plan . . . analyze . . . work forward . . . work backward . . . imagine . . . organize . . . use analogies . . . brainstorm . . . proceed by trial and error . . . select . . . the list goes on and on.

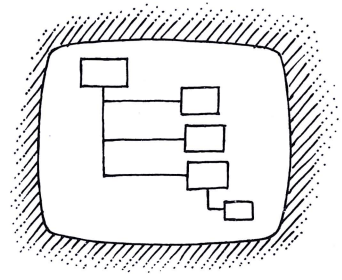
So far, human experience tells us that there is no one best way to solve a problem. The same is true with writing a computer program—a variety of lines of code can produce the same effect. What has been found, however,





STAGE 1- EXPLORE WITH TURTLE GRAPHICS.

STAGE 2- LEARN LOGO'S PRIMITIVES,
WRITE SIMPLE PROGRAMS.



STAGE 3- USE TOP-DOWN PROGRAM DESIGN.

Fig. 5-1. Three stages of programmer development.

is that the best approach is the "top-down" design or the "structured programming" method of addressing the problem.

What exactly do we mean by top-down or structured programming? We'll use a method we've devised, called the "box chart," to guide our efforts. The box chart is a paper-and-pencil way to plan a program. Suppose we want to draw a picture of a house using turtle graphics. Let's assume that we want a house of varying size. The top-down approach begins by planning our HOUSE program from the "top" and adding details as we go. (By "top-down" we don't mean that we draw the roof first!)

First of all, here is a rough picture of what we want to draw.

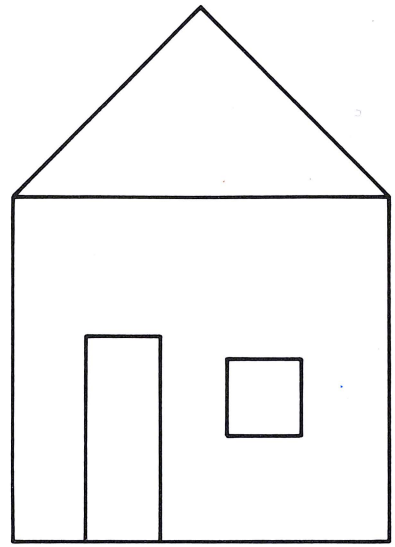
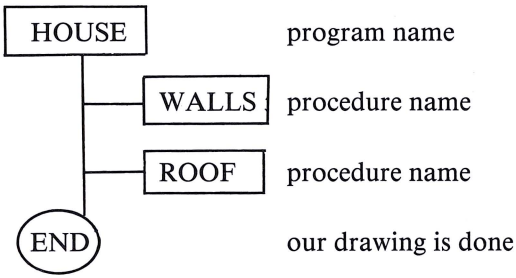
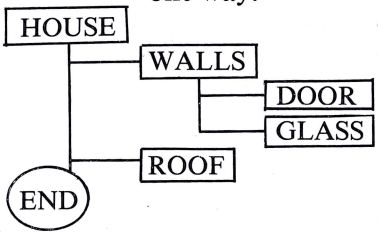


Fig. 5-2. House.

The box chart plan might look like this:

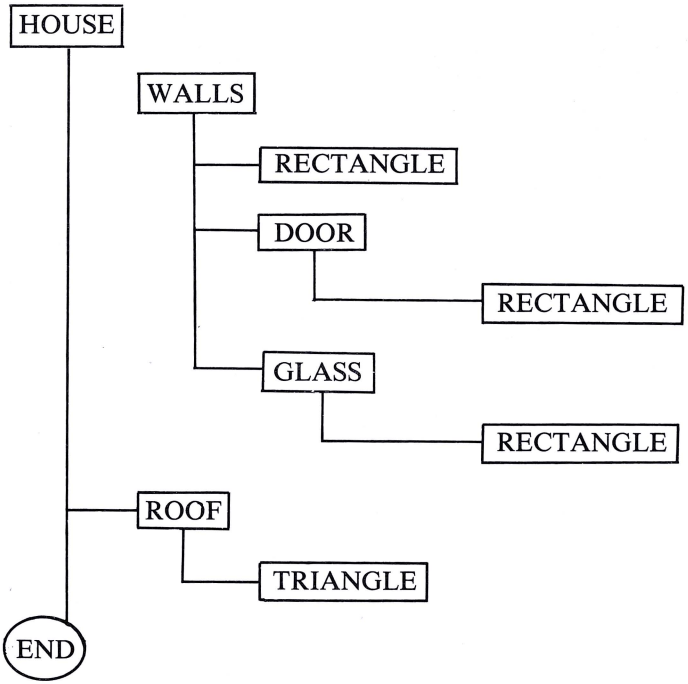


In the box chart, we try to take the big program and divide it up into natural modules. A house, after all, can be thought of as a structure made of walls and a roof. Now let's add some more refinements. Suppose we want to add a door and a window to the house. How do we map out this? Here's one way:



we add two more procedures to our chart
we can't call this WINDOW; WINDOW is a primitive

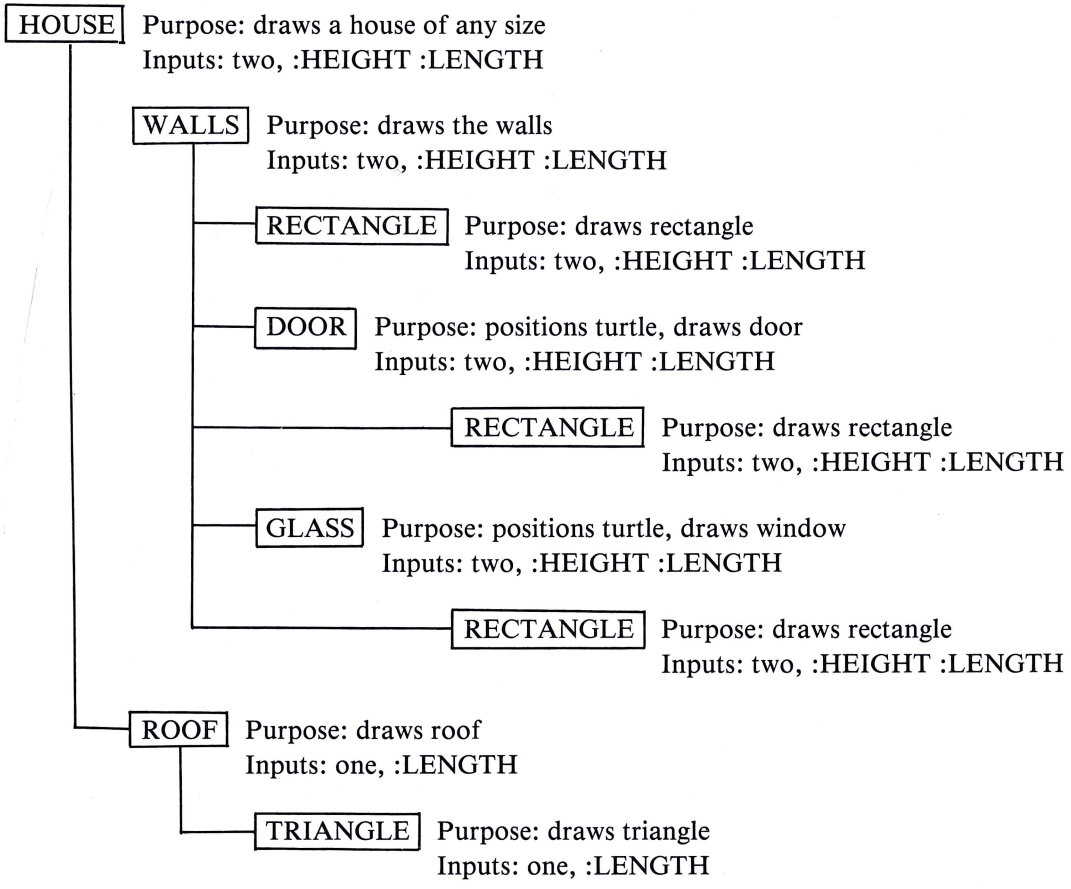
As we think about it, we realize that the walls, door, and glass window could all be drawn using a RECTANGLE utility. And, our roof can be drawn by placing a TRIANGLE on top of the rectangular walls. We can refine our box chart to reflect this:



Thus, our approach to "top-down program design" is to plan the program from the top adding refinements and details at lower levels. Each box in the chart would be a procedure. Primitives are not shown. We've found

this approach very useful. If you flip through this book you'll see dozens of box charts that have evolved like this to their final state.

At this point, you might feel like you're ready to "whip out" the details and write each procedure. That's fine, but we'd like to suggest one more planning step. (Actually this step can be done now, or in the middle of writing your procedures, or at the end. We'll do it now.) Let's map out what each boxed procedure will do. This documentation explains what each boxed procedure does and what inputs it takes. Since we'd like to construct houses of varying heights and lengths, we'll want to include this information in the box chart. In order for everything to be proportional, we'll need inputs for :HEIGHT and :LENGTH in most of the procedures we write. One exception is for the ROOF, which will only depend on the building's length. Our expanded box chart includes the additional information, along with a brief description of the purpose of each procedure.



Now that we have our box chart map in front of us, let's actually write the program code for constructing a house. As we've outlined in the chart above, the program HOUSE will call WALLS AND ROOF.

```
?TO HOUSE :HEIGHT :LENGTH
>WALLS :HEIGHT :LENGTH
>ROOF :LENGTH
>END
```

We can now begin writing the subprocedures themselves. Here is a listing for WALLS.

```
?TO WALLS :HEIGHT :LENGTH
>RECTANGLE :HEIGHT :LENGTH
>DOOR :HEIGHT :LENGTH
>GLASS :HEIGHT :LENGTH
>END
```

Notice how WALLS just calls other procedures. Once again, we push the details down to subprocedures whenever possible. The RECTANGLE procedure is our utility from the last chapter.

According to our box chart, the next stage of construction calls for us to draw in a door. WALLS left the turtle at the lower left corner of the house, pointed in the "Y" axis. We first need to rotate the turtle through 90 degrees, and then move it down the length or "X" axis of the house. Then, we rotate the turtle left 90 degrees, and draw a rectangle which will be the door.

```
?TO DOOR :HEIGHT :LENGTH
>RIGHT 90                                  Position turtle
>FORWARD :LENGTH /4
>LT 90
>RECTANGLE :HEIGHT/1.3 :LENGTH /4          Draw door
>LT 90                                     Put turtle back where
>FORWARD :LENGTH /4                        we found it
>RIGHT 90
>END
```

DOOR calls RECTANGLE. Do we need to write another RECTANGLE procedure? No, Logo is able to use the utility written earlier. All we needed to do in DOOR was to divide the current input values for the rectangle's height and length to make our door a proportional size to the house.

Notice how we moved the turtle back to the starting position. We do this in order to keep the DOOR procedure as "independent" as possible. We can remove the DOOR procedure or replace it with another version without affecting the rest of the program.

The next procedure is GLASS. As with DOOR, we'll need to move the turtle into position, and then call a RECTANGLE of suitable size. After drawing the window, we bring the turtle back to the starting point.

```
?TO GLASS :HEIGHT :LENGTH
>RT 90                                  Position turtle
>FD :LENGTH/1.7
>PU
>LT 90
>FD :HEIGHT/3
>PD
>RECTANGLE :HEIGHT/4 :LENGTH/4 Draw window
>PU
>RT 180                                 Put turtle back
>FD :HEIGHT/3
>RT 90
>FD :LENGTH/1.7
>RT 90
>PD
>END
```

The HOUSE's WALLS, DOOR, and GLASS are now completed. The only procedure yet to be written, according to our box chart, is the ROOF. An "A" shaded roof can be a triangle placed on top of the walls. We start by positioning the turtle.

```
?TO ROOF :LENGTH
>FD :HEIGHT
>TRIANGLE :LENGTH
>BK :HEIGHT
>END
```

Notice that ROOF takes only one input, which is how we planned it in our original box chart. Now we find that we also need to use the variable :HEIGHT to position the turtle. Is that okay? Can we use :HEIGHT without naming it as an input to ROOF? Yes, as long as :HEIGHT was called in the HOUSE procedure, its value can be used in subprocedures. We could

have put :HEIGHT in the title line of the ROOF subprocedure and that would have worked fine, too. In Chapter 9, we'll discuss inputs in detail.

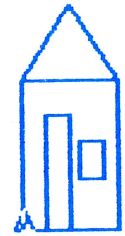
ROOF calls the utility TRIANGLE. Here is a listing.

```
?TO TRIANGLE :SIDE
>RT 30
>REPEAT 3 [FD :SIDE RT 120]
>LT 30
>END
```

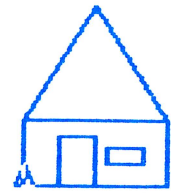
You might note a tricky sleight-of-hand here. TRIANGLE has :SIDE for its input, while ROOF calls TRIANGLE WITH :LENGTH as input. Will this work? Yes, recall that dotted words represent a value, in this case, a number. So TRIANGLE just gets its number input from LENGTH. In the procedure TRIANGLE, :SIDE is a "dummy" variable that serves as a generic name for TRIANGLE's input. The use of :SIDE creates a variable name. The variable value comes from the ROOF procedure.

We can now run the program by calling HOUSE and providing Logo with the numeric inputs it needs. You could, for example,

```
?HOUSE 70 40
```



```
?CLEARSCREEN
?HOUSE 30 60
```

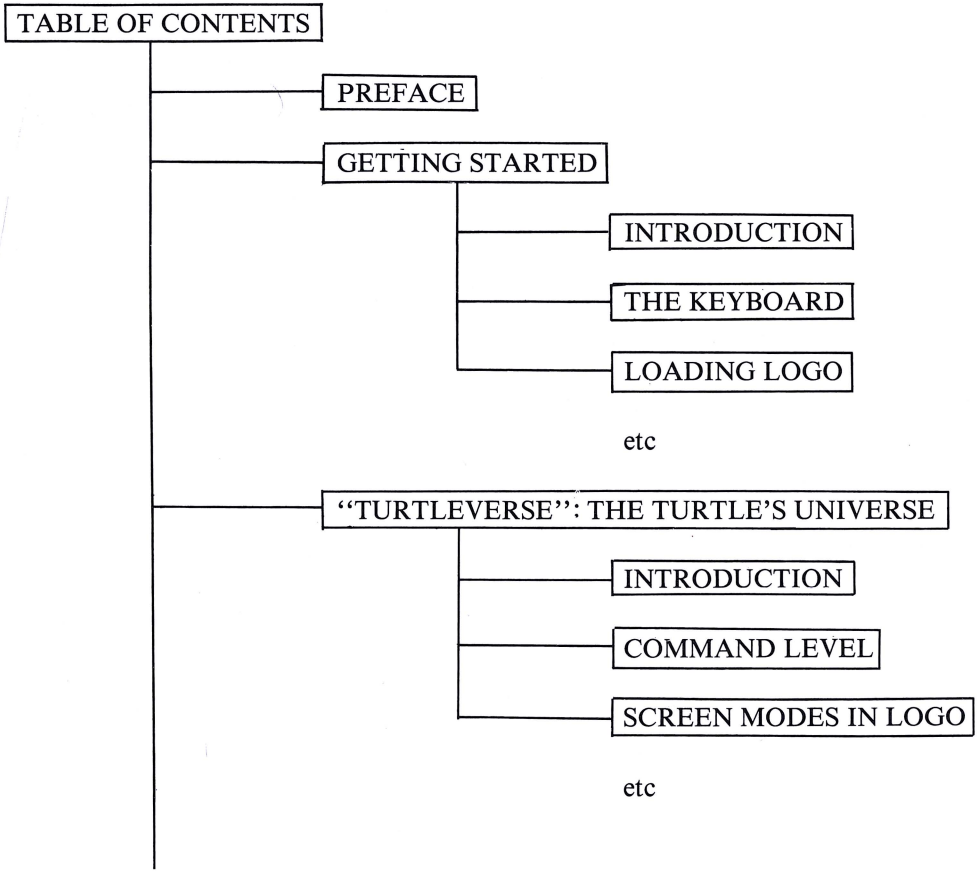


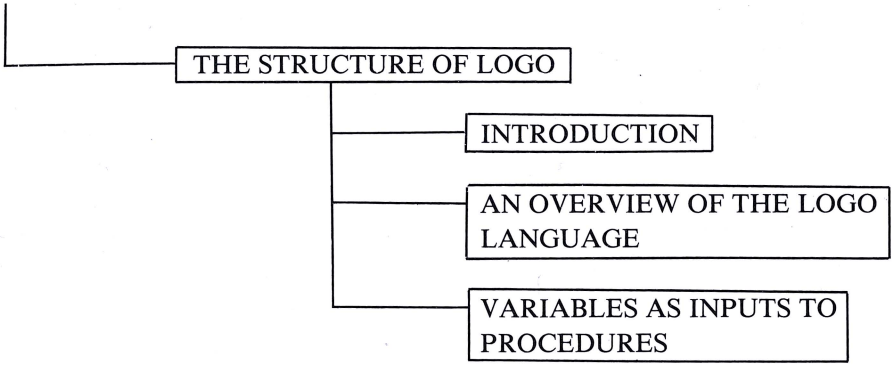
We now have a HOUSE program that we can add to our graphics library.

With simple procedures like HOUSE, it is easy to still use trial and error and end up with a nice house. However, suppose you wanted to build a "real" house. You could go to the lumber yard, buy some tools and lumber, deliver it to your lot, and start hammering away. This method would work, but almost everyone, including experienced carpenters, would rather work from a set of architect's plans. Complex programs are like complex buildings. They should be planned out first.

5.3 Using the Box Chart Method

One can think of the box chart approach as being similar to the organization of a book, or a term paper. For example, we could box chart the first three chapters of this book.





etc

Many of the characteristics of good writing and organization also apply to programming. Each chapter in a book should stand by itself as much as possible. Only major, essential ideas should be passed on to the next chapter. Each chapter should start out with an overview, relegating details to specific subsections.

The same is true for programs. A good program should be broken up into "modules" or procedures." Each procedure should be as independent as possible. Program details should be deferred to subprocedures. This approach provides the best chance for writing good procedures.

Here is a summary of the advantages of the box chart approach.

1. Programs are easier to plan.
Box charts provide an overview of what we are trying to do. The details (how each box works on the inside) are postponed to later.

2. Programs are easier to debug.
Each box or procedure can and should be tested by itself before making it a part of a larger program.

3. Programs are better documented.
The essential information about each box or procedure is provided. Our full box chart indicates what each procedure does, and what information it needs.

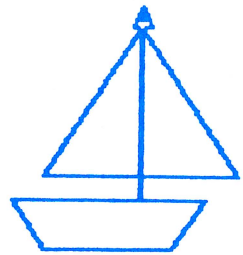
4. Programs are easier to read.
First we look at the box chart to see what the overall structure is, and then we look at the details of how each part works.

5. Programs are easier to change.
We can remove a box or procedure and substitute another version with the same connections to the other procedures. Or, we can add boxes containing refinements to each level. For example, you could add a pane of glass to the door by adding the procedure called DOOR.GLASS in our HOUSE design earlier.

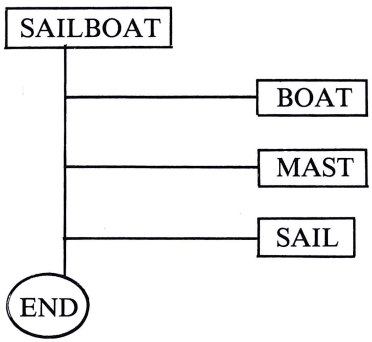
We think you'll find using the box chart approach helpful in designing both simple and complex projects. In the next section, we'll suggest some programming projects to practice top-down planning.

5.4 Programming Projects Using the Top-Down Approach

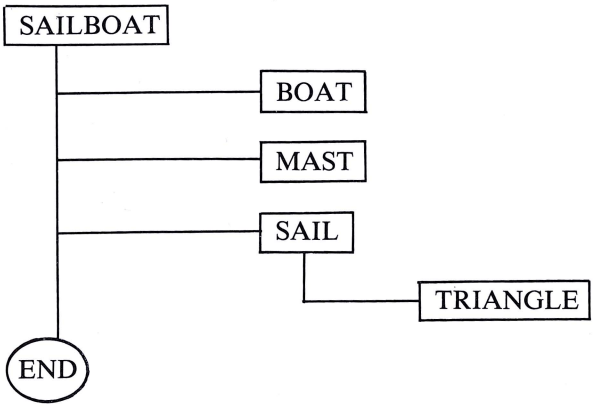
The second example of a turtle graphics project is the design of a sea of sailboats floating across the video screen. We begin by planning how to construct a sailboat which could be placed at any location on the screen. The sailboat might look like this:



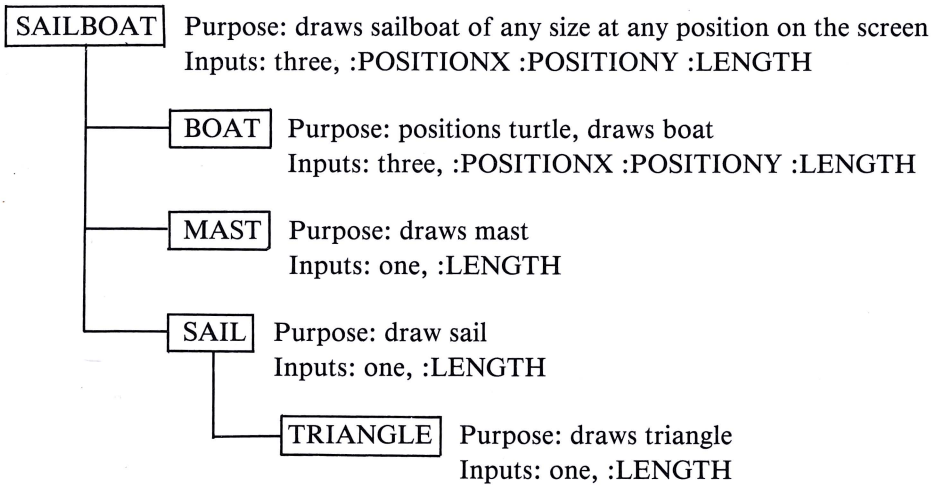
Begin the project by box charting the sailboat, as was done with HOUSE.



Notice the familiar TRIANGLE shape in the sail. We can use TRIANGLE in GRAPHICS for this part of the drawing.



Before writing the program, let's include the purpose and inputs required for each of the procedures.



Now write the actual procedures.

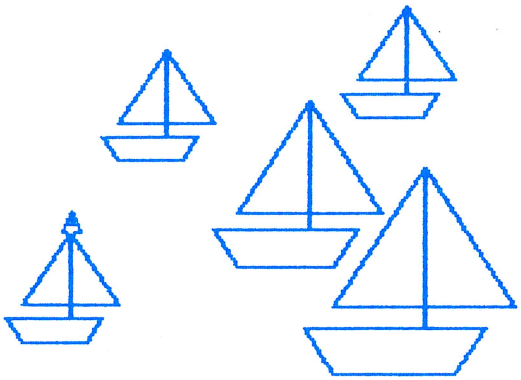
```
?TO SAILBOAT :POSITIONX :POSITIONY :LENGTH
>BOAT :POSITIONX :POSITIONY :LENGTH
>MAST :LENGTH
>SAIL :LENGTH
>END

?TO BOAT :POSITIONX :POSITIONY :LENGTH
>PU
```

```
>SETX :POSITIONX
>SETY :POSITIONY
>PD
>LT 30
>FD :LENGTH / 3
>RT 120
>FD :LENGTH
>RT 120
>FD :LENGTH / 3
>SETX :POSITIONX          Direct the turtle back to the starting position
>RT 120
>END

?TO MAST :LENGTH
>FD :LENGTH / 3
>RT 120
>FD :LENGTH * 2 / 3
>LT 90
>FD :LENGTH
>END

?TO SAIL :LENGTH
>FD :LENGTH
>RT 120
>TRIANGLE :LENGTH
>LT 120
>END
```

To draw the flotilla, call SAILBOAT and provide it with inputs for the X and Y positions and the length. The drawing at the top of the next page was created using these inputs:

```
?SAILBOAT 30 -60 75
?SAILBOAT 40 60 40
?SAILBOAT -10 -10 60
?SAILBOAT -60 40 40
?SAILBOAT -100 -45 40
```

Here are a variety of other project ideas for you to plan and draw in turtle graphics. As you look at the figures, think about the approach we used in writing our HOUSE and SAILBOAT programs. Begin by breaking each picture down into its natural components, box charting each subprocedure,



and then writing the lines of code. Try to push the details down to lower levels and try to make each procedure (box) as independent as possible.

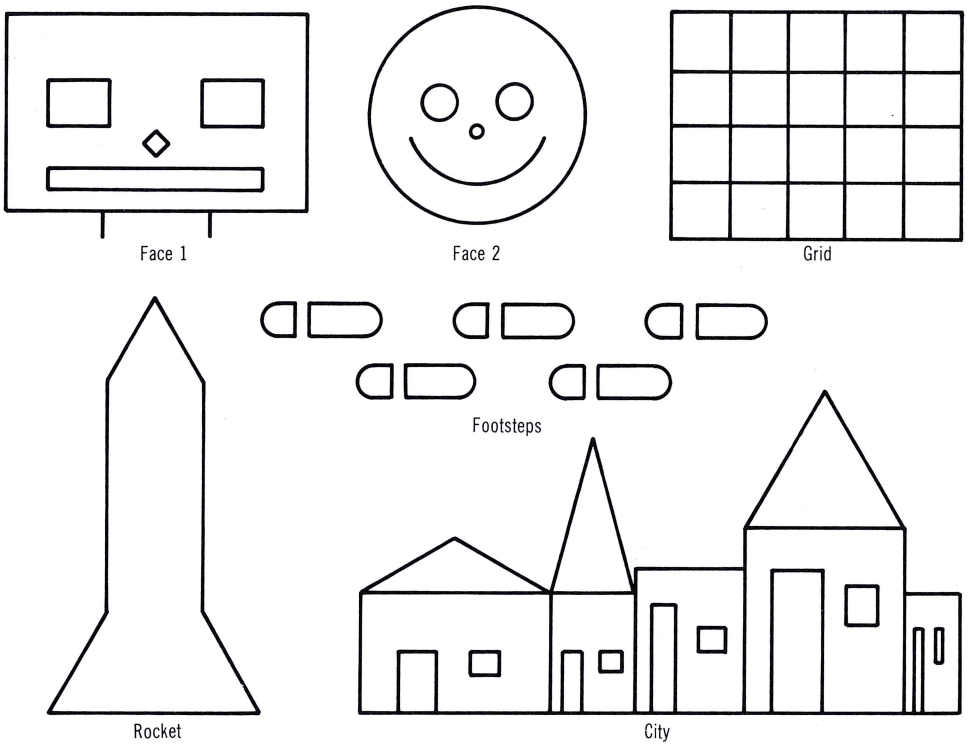


Fig. 5-3A. Programming projects.

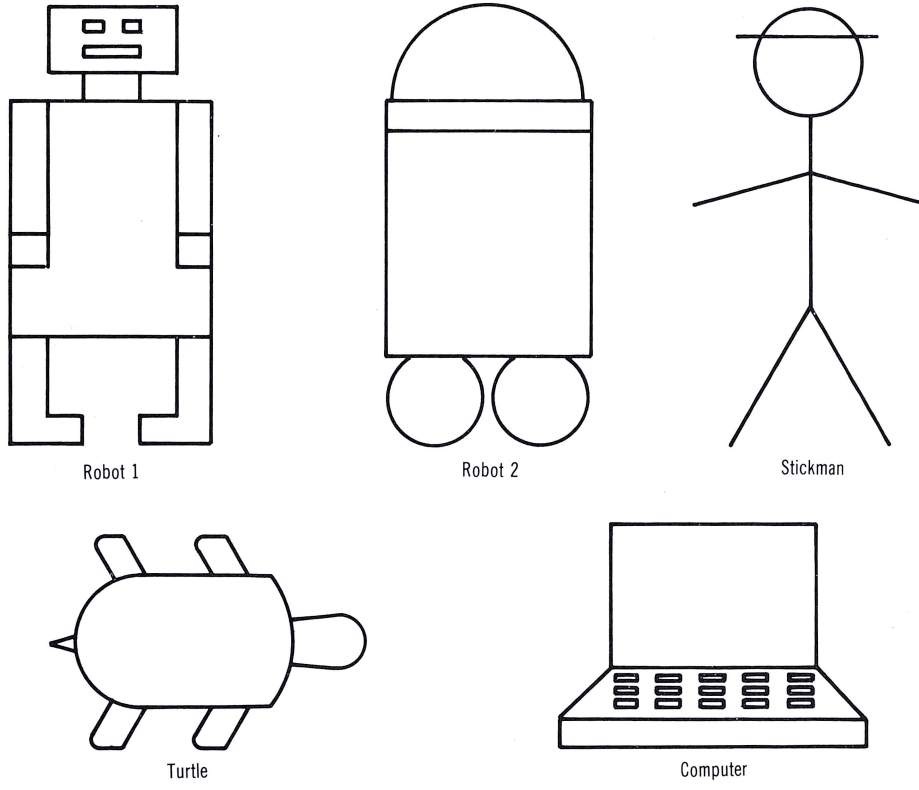


Fig. 5-3B. Programming projects.

5.5 Program Control

In the discussion of Logo's structure in Chapter 3, we saw that each line in a procedure begins with a command (or procedure call) that may or may not take an input. Each line of the procedure is "executed" in turn. In addition, when a procedure "calls" another procedure, that procedure runs until it ENDs. Control is then passed back to the next line of the original or "calling" procedure. To illustrate this, we'll modify the SAILBOAT box chart in Fig. 5-4 to show the starting and stopping elements.

The program runs by directing Logo to SAILBOAT followed by three inputs. SAILBOAT calls on BOAT, which draws a raft and ENDs, passing control back to SAILBOAT. SAILBOAT proceeds by calling the next procedure, MAST, which draws the pole for the sail and then ENDs. Control is

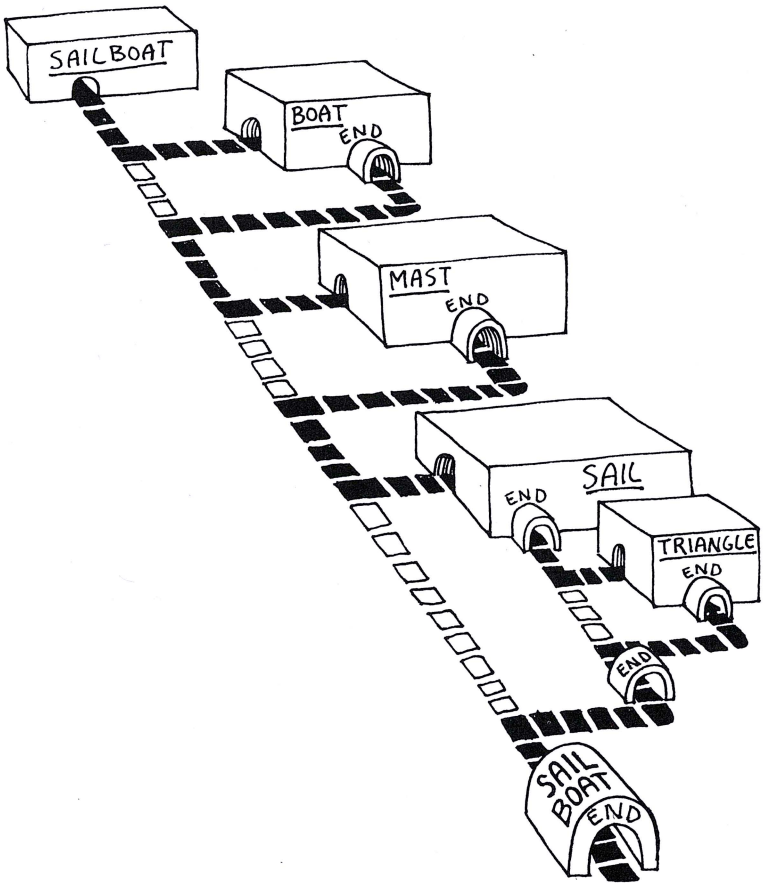


Fig. 5-4. Program control in SAILBOAT.

again passed to SAILBOAT which then calls SAIL. SAIL calls TRIANGLE; TRIANGLE ENDs and then SAIL ENDs. When SAIL ENDs, control is passed back to SAILBOAT which executes its last call, and ENDs.

When do procedures end? Or better yet how do they end? It turns out there are five ways to end a procedure (or six, if you include turning off the Apple . . .). Three of the ways are "natural" in that we can include them as part of the program; two ways are "unnatural" or abrupt and usually occur if there is a program "bug."

The natural ways to end a procedure are with the commands STOP and END or with the operation OUTPUT. You're familiar with END as we've used it to end all of our procedures and programs. A program ends when

END is encountered. In Chapter 6 we'll define STOP; OUTPUT is discussed in Chapter 8.

The abrupt ways a procedure quits or "crashes" are with an error message from Logo, or with a keyboard control sequence called CTRL-G.

Error messages from Logo indicate the source of the problem; after you correct the problem, you can run the procedure again by typing its name. If, for example, you tried to run SAILBOAT before you had written MAST, Logo would draw the boat hull and then display the following:

```
I DON'T KNOW HOW TO MAST IN SAILBOAT:
MAST :LENGTH
?□
```

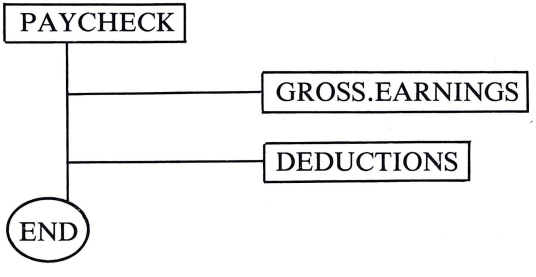
The other way a procedure stops abruptly is with the keyboard sequence CTRL-G. CTRL-G is an "emergency" stop which we'll discuss further in Chapter 6.

5.6 Another Top-Down Planning Example: The Paycheck

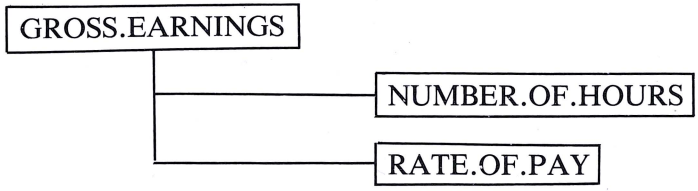
Here is an example of top-down planning involving the production of a payroll ledger. We won't complete the process by actually writing each procedure (we need to know a few more primitives first); we'll focus instead on the approach. As the program unfolds, notice how we think only about the next step, then draw the boxes which help us visualize the next step, and so on.

Let's begin by identifying the major components of the solution to the problem. For our problem, the question asked is, "What is the amount to be written on an individual's paycheck?"

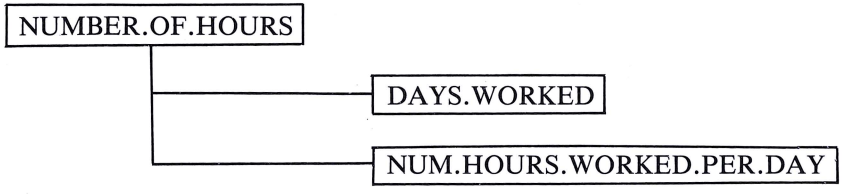
From the broadest view, a person's monthly paycheck is the difference between the gross amount earned and any deductions. Our box chart would reflect that initial distinction:



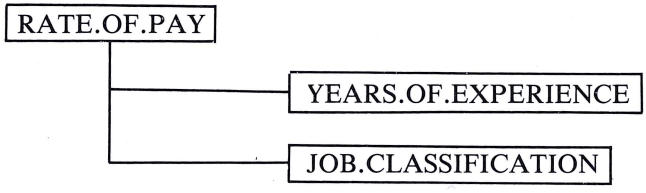
Now let's refine our steps. Gross earnings are usually determined by the number of hours worked times the rate of pay. We can refine our gross earnings procedure to reflect that:



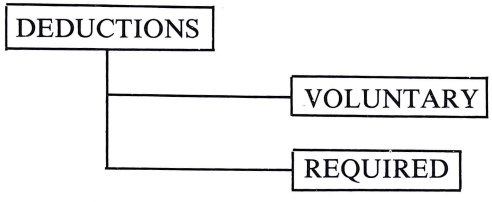
The number of hours is often calculated by subtracting days absent from the days in a month, and then multiplying that number by the hours worked per day:



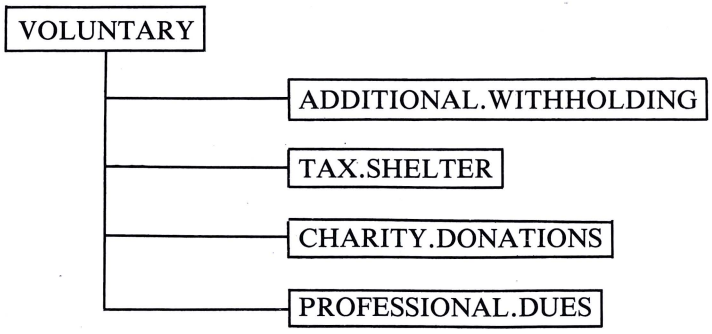
The rate of pay is frequently determined by years of experience and job classification:



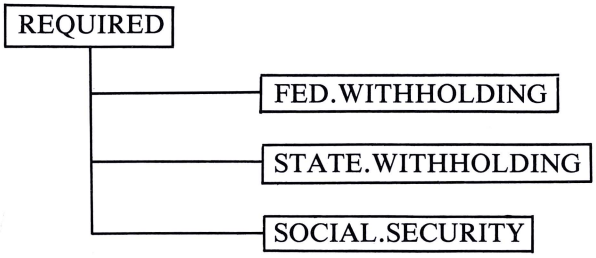
In the same way, one can see deductions as typically being either "voluntary" or "required":



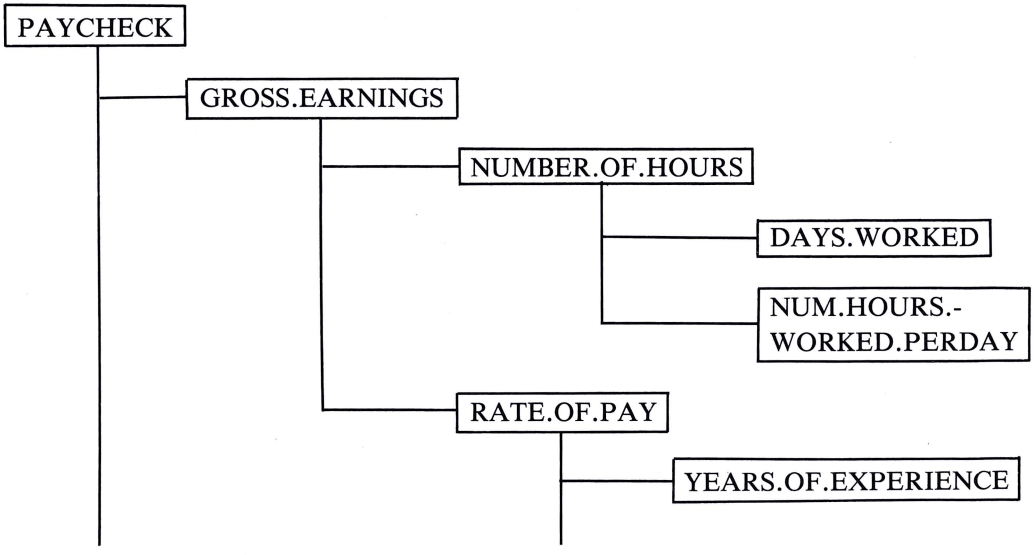
Voluntary deductions could consist of several types, including the following:

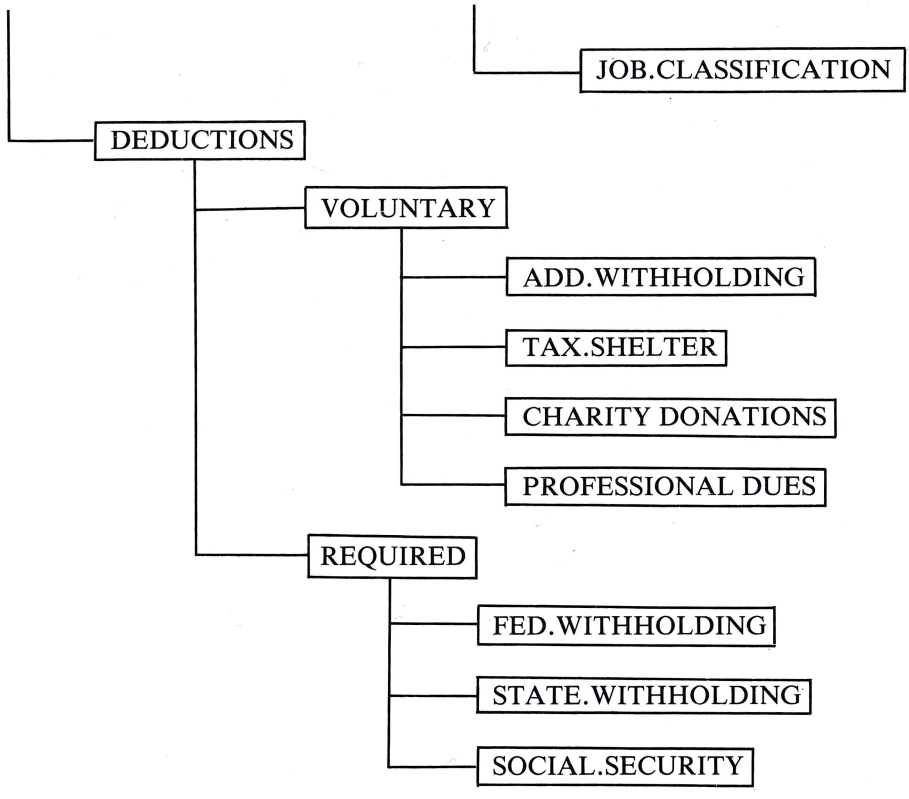


Required deductions could also be of several types, depending on the situation:



Our PAYCHECK program, then, has been refined to show the following components:





When we started the PAYCHECK program, you might have thought "This is easy; there are just a couple of steps and that's it." However, a computer needs very detailed instructions and the best way to provide that information is to plan from the top-down.

Each box may or may not be a procedure. When we start to write the "top level" procedures first, we may find we can combine two or more "boxes" together. However, our top-down design clearly spells out the major components of the problem.

Again, we won't actually write the Logo program for PAYCHECK. Our point here is that this type of problem, like those we solved involving turtle graphics, can be approached from a top-down design, adding refinements as we go.

5.7 Summary

Here are some important points we've covered in this chapter:

1. As individuals learn Logo, they usually go through three recognized stages of development.
2. The difference between a master programmer and a novice is more than just the number of primitives the former knows; instead, it is that individual's orientation to problem solving using Logo.
3. Although there are many approaches to problem solving using a computer language, we've found that top-down, structured programming is the most efficient and effective.
4. Our use of box charts codifies and clarifies this approach. We begin by breaking the task into major subunits, and then adding stepwise refinements.

5.8 Review Questions and Exercises

Review Questions

1. What is the difference in approach to solving a programming problem between a beginner (stage 1) and a master (stage 3) Logo programmer?
2. What do we mean by structured or top-down programming?
3. What is the relationship between box charts and structured programming?
4. Why do we include "inputs" in our box charts?
5. Look at the box chart for HOUSE and decide how you would modify the program to give the dwelling a flat instead of a triangular roof.

Exercises at the Terminal

Be sure to try at least two of the programming projects in Fig. 5-3; we'd recommend you box chart several others!

Answers to Review Questions

(1) At stage 1, problems are solved by a trial and error approach; at stage 3 the programmer maps out the approach first. (2) By structured, we mean that the programming problem is analyzed by breaking the task down into natural subcomponents. Then each subcomponent is further analyzed and

broken down into its component parts. It is the opposite of "bottom up," "let's see if this works" programming. (3) Box charts are an effective method of planning our programming. They are a natural outcome of the structure approach, as they provide us with a visual representation of the problem and its components. They are a way of seeing the forest and the trees at the same time! (4) The inputs remind us what values each subprocedure will need in order to run. (5) First, modify HOUSE so that ROOF takes two inputs: HEIGHT and LENGTH. Then, rewrite ROOF so that a flat roof would be drawn. Here is one example:

```
?TO ROOF :HEIGHT :LENGTH
>LT 90
>FD :LENGTH/4
>RT 90
>RECTANGLE :HEIGHT/10 :LENGTH*1.5
>END
```

CHAPTER 6

RECURSION

CONCEPTS:

simple recursion
using IF—condition true—STOP
complex recursion
recursive analogies
using recursion to solve problems

PRIMITIVES:

IF, STOP

6.0 Introduction

Think about the many tasks that we perform that involve repetitive sequences of searching, sorting, adding, counting, itemizing, etc. Activities as varied as developing a grocery list, balancing a checkbook, and filling out income tax returns all involve discrete steps and procedures that once completed, are started all over again.

For example, consider the steps involved in preparing a weekly shopping list:

1. Check for staples (coffee, milk, flour, bread, etc.), and note those that are in short supply.
2. Prepare weekly menus; list ingredients, check pantry and freezer to see what items are needed.

3. Check calendar to see if there are any special occasions (birthdays, anniversaries, etc.) that require preparation of special foods; check for needed ingredients and "props" such as candles, etc.

4. Next week, repeat the entire process . . .

Although all of us enjoy the finished product (good meals!), many of the steps in our grocery list analogy are not very exciting and quite frankly take a lot of time. Computers have it over on humans in cases like this: computers love repetition, and to be honest they do it faster and do it more accurately then we do.

One of Logo's powerful features is its ability to do repetitive tasks easily. The process by which this is done is called "recursion"; programs that use it are said to be "recursive." Just about every program in this book uses recursion. When we want Logo to search, to count, to sort, or to produce intricate drawings and graphics, we use the power of recursion.

Recursion, briefly put, is writing procedures that call themselves. Although its effect is often subtle and complex, at the simplest level we can describe it as a repetitive process. For example, in the grocery list analogy above, the Logo version might look like this:

```
?TO GROCERY.LIST
>STAPLE.CHECK
>MENU.CHECK
>EVENT.CHECK
>WAIT ONE.WEEK
>GROCERY.LIST      The "recursive call"
>END
```

Recursion occurs in the next to last line; by including GROCERY.LIST as part of our definition of GROCERY.LIST, we've defined a procedure that will repeat the process of GROCERY LISTing. If this process of defining a procedure in terms of itself seems to be circular (as in "all tautologies are tautologies"), you're right. But Logo isn't bothered by this rhetorical problem; it happily accepts definitions which include themselves in the definition.

Again, recursive procedures are procedures that call themselves. When the "recursive call" comes at the end of a procedure, the recursive action can be seen as a command to repeat, as we saw in the GROCERY.LIST analogy. If it is placed in a different location, its effect is much more complex, as we will see later.

In this chapter, we'll begin by examining the simplest cases of recursion,

and then look at the full power of the process. Although the concept of recursion is abstract, we'll look at numerous examples and analogies in order to develop an intuitive understanding. In a short time, you will find yourself using recursion to get Logo to do some pretty amazing things.

6.1 Simple Recursion: Four Examples

We'll begin by looking at four examples of simple recursion.

6.1.1 Recursion with Text

Let's say we wanted to write a procedure that would direct Logo to repeat a PRINT command. We could do this several ways. One way would be to use the REPEAT command, along with an input, as in the following:

```
?TO TALKALOT :X
>REPEAT :X [PRINT "HELLO]
>END
```

We could then direct Logo to TALKALOT 20, or TALKALOT 30. If we did, Logo would respond by printing "HELLO" 20 or 30 times, respectively.

Another way would be to use recursion, i.e. to have the procedure TALKALOT call itself. The recursive procedure would look like this:

```
?TO TALKALOT
>PRINT "HELLO
>TALKALOT      Recursive line; directs LOGO to call or
>END            Run TALKALOT again
```

The procedure begins by informing Logo that the first thing to do in TALKALOT is to print "HELLO." The next line of the program, the recursive call, tells Logo to TALKALOT. Logo prints "HELLO," and then again calls TALKALOT, which responds by printing "HELLO" again and so forth. We call this "looping." If you were to run the program, the screen would look like this:

```
HELLO
HELLO
HELLO
HELLO
```

The bottom HELLO could be seen to be faintly flashing; each time a new HELLO is printed, the top HELLO goes off the top of the screen, and the new one scrolls up. This continuous printing of "HELLO" will go on ad infinitum—our program is looping back again and again to print "HELLO." There is an "emergency stop" if you find yourself in such an "infinite loop." While holding the CTRL or CONTROL key down, hit the letter "G." This operation, called "CTRL-G" will stop the program. The familiar ? and flashing cursor will be at the bottom of the screen, signaling that you are in control again. Later in this section, we'll see how to write recursive procedures with a "stop" as part of the procedure.

6.1.2 Recursion with Graphics

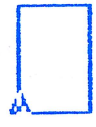
Our second example of simple recursion involves graphics. Suppose we had written a procedure to draw a box shape:

```
?TO BOX
>REPEAT 2 [FD 50 RT 90 FD 30 RT 90]
>END
```

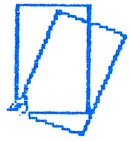
Let's create a design by drawing BOX, rotating the turtle head through twenty degrees, and drawing BOX again, rotating another twenty degrees, etc. As with TALKALOT, the REPEAT command could be used. But here is a recursive version:

```
?TO RIGHTBOX
>BOX            draws a box
>RT 20          rotates turtle 20 degrees right
>RIGHTBOX       calls RIGHTBOX again
>END
```

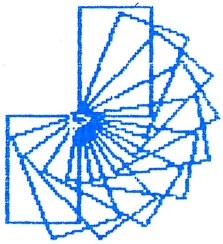
When you finish typing in the program, call RIGHTBOX. As the picture unfolds on your screen, analyze what is happening in terms of the procedure we wrote:



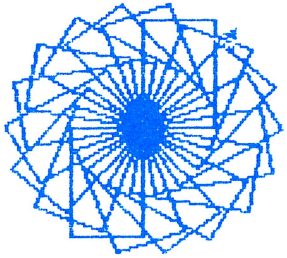
Logo draws a box and rotates the turtle's head 20 degrees



The recursive call "RIGHTBOX" directs Logo to again draw a box, and rotate 20 degrees



Logo has gone through ten such RIGHTBOXes at this point

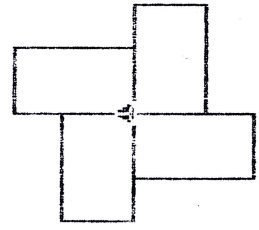


A pattern is "finished;" on your screen you'll see the turtle draw over the same lines again and again. To stop it, use CTRL -G

Earlier we wrote procedures using variables as inputs. Let's incorporate inputs into our RIGHTBOX program. Let's write a program that will allow us to vary the angle of the box's rotation from its current 20 to whatever value we like. Edit RIGHTBOX so that it reads as follows:

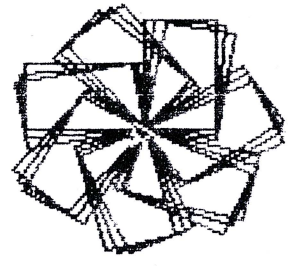
```
?TO RIGHTBOX :ANGLE      Add the variable "angle"
>BOX
>RT :ANGLE               Indicate where the variable is used
>RIGHTBOX :ANGLE         Include the variable in the recursive line
>END
```

Now run RIGHTBOX with different values for the angle. If you RIGHTBOX 90, for example, you'll obtain a pattern like this:



Logo draws the box, turns 90 degrees, and makes the recursive call. To stop the pattern, use CTRL-G.

Now try RIGHTBOX 155:



Logo draws the box, turns 155 degrees, draws the box, turns 155 degrees, etc.

Try using other inputs for RIGHTBOX :ANGLE. What pattern, for example, do you get with RIGHTBOX 180? RIGHTBOX 175?

Our second example of a recursive procedure using graphics and inputs is called SPIRAL. The procedure for SPIRAL is as follows:

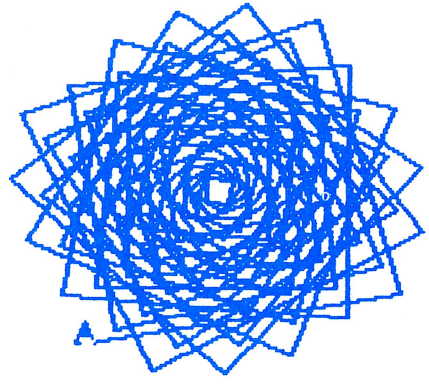
```
?TO SPIRAL :DISTANCE
>FD :DISTANCE
>RT 95
>SPIRAL :DISTANCE + 1
>END
```

This procedure is different from RIGHTBOX in that the recursive call, SPIRAL :DISTANCE + 1, has a new input value. Each time Logo encounters this procedure line, it adds 1 turtle step to the distance it moves forward. Thus, if we command Logo to

```
?SPIRAL 20
```

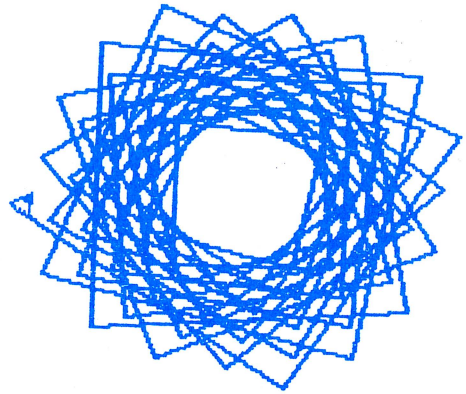
the turtle will go forward 20 steps, turn right 95 degrees, then go forward 21 steps (distance + 1 = 20 + 1 = 21). Because the procedure is recursive, the turtle will again turn 95 degrees right, and then go forward 22 steps (21 + 1 = 22). When the procedure has run 10 times, the turtle will be moving 30 steps FORWARD before turning right 95 degrees.

Let's run some SPIRAL :DISTANCE procedures. If you command Logo to SPIRAL 10, the screen will display the following:



SPIRAL 60 will give you the pattern shown at the top of the next page.

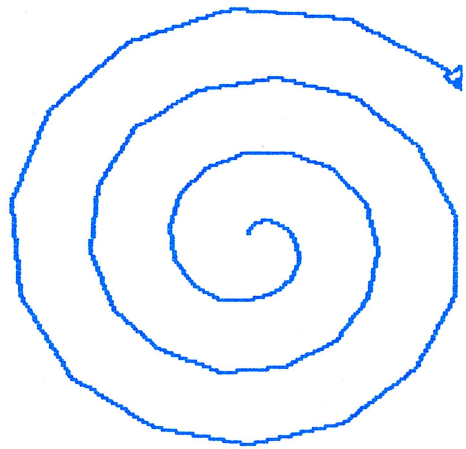
Let's modify SPIRAL :DISTANCE to include a second variable. Instead of always having Logo turn 95 degrees to the right, let's make the degree of rotation a variable along with the distance FORWARD. Edit SPIRAL to read as follows:



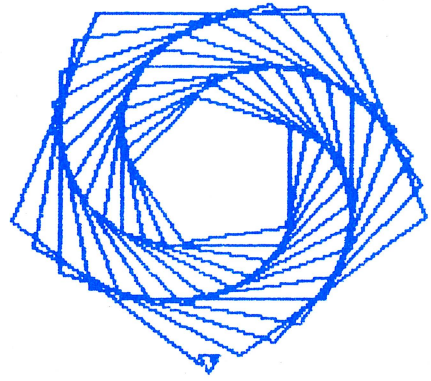
```
?TO SPIRAL :DISTANCE :ANGLE
>FD :DISTANCE
>RT :ANGLE
>SPIRAL :DISTANCE +1 :ANGLE
>END
```

Here are some examples:

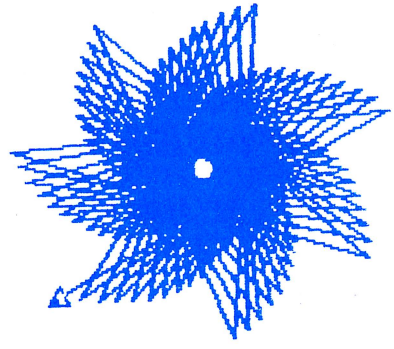
```
?SPIRAL 1 25
```



```
?SPIRAL 40 290
```



```
?SPIRAL 40 155
```



These are fun to play with. Can you find any "general rules" for inputs that give the most pleasing designs?

6.1.3 Use of the IF—Condition—STOP Command in Recursion

Our last example of simple recursion will include a way to prevent the infinite looping in recursion. Suppose we want to print a sequence of integers: 1, 2, 3, etc. Essentially, we want Logo to "count up" to a given number, say "4," beginning with some initial number, say "1." We'd begin our definition of COUNT.UP by designating the variable in the first line of the program, as we've done before:

```
?TO COUNT.UP :X
>IF :X=5 [STOP]
>PRINT :X
>COUNT.UP :X+1
>END
```

Let's take a brief look at the second line of the procedure. This line controls the amount of repetition or recursion. We do this by using the Logo primitive IF. The primitive IF requires at least two inputs as follows:

IF some condition is true, then do this

Input #1 Input #2

As we'll see in Chapter 9, when we give a formal definition of IF, we can have a third input for IF. For now, though, we'll use IF followed by just two inputs. The format for IF is as follows: type IF, followed by the condition you wish to check, followed by the action you wish performed, if true, in brackets. Here are some examples of IF statements:

```
IF :A = :B [PRINT "BALANCED]

IF :ANSWER > 50 [PRINT [YOU WIN!]]

IF XPOS = 20 [FD 40 RT 90]

IF :X = 5 [STOP]
```

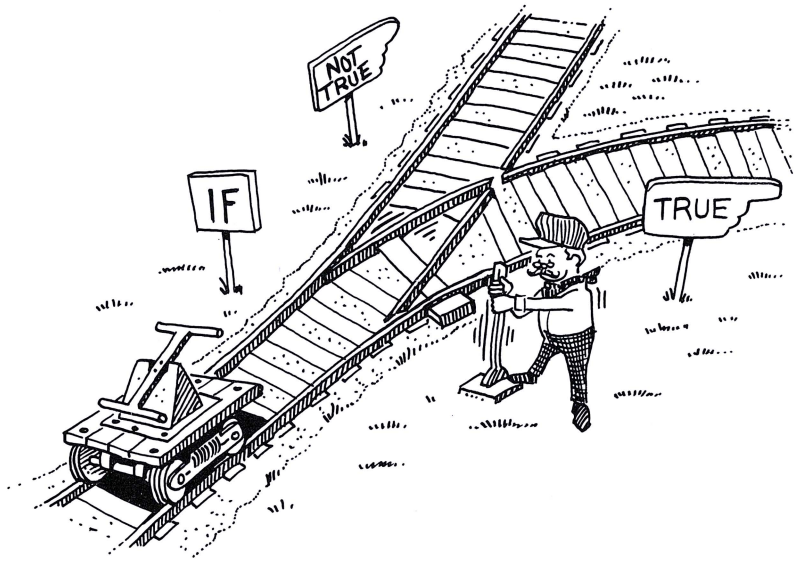


Fig. 6-1. IF—Condition—Action.

The last example, IF :X=5 [STOP] introduces the Logo primitive, STOP. STOP does just that; if, in the example above, :X does equal five, the program terminates. Aha! A way to handle the infinite loop problem in recursion. We decide in advance how many times to repeat a process, and write a conditional IF statement to STOP it. Suppose that in the COUNT.UP procedure, we decided to "count up" to the number four. Our conditional IF statement, then, would look like this:

```
>IF :X = 5 [STOP]
```

The third line of COUNT.UP directs Logo to print the procedure input:

```
>PRINT :X
```

Notice that STOP is executed before the PRINT command. If we moved the conditional line IF :X = 5 [STOP] after the PRINT command, we would have directed Logo to PRINT 5 before it STOPped.

The recursive call in the COUNT.UP procedure directs Logo to run a new procedure (called COUNT.UP) and give it the input value of :X+1.

```
>COUNT.UP :X + 1
```

Test the program by directing Logo to COUNT.UP with an initial input of 1 as shown below:

```
?COUNT.UP 1
1
2
3
4
?
```

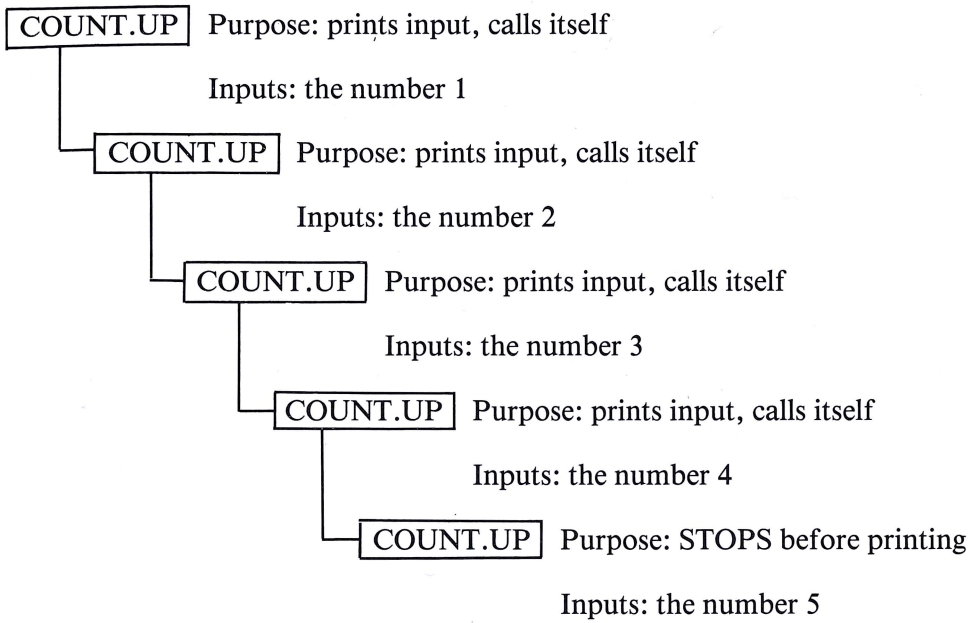
6.1.4 Analyzing the Recursive Process

Let's examine the COUNT.UP procedure more closely to see how it works. We'll analyze it in terms of the box chart that we've used earlier. To the right of each box we list the purpose and inputs for that procedure.

Here is a listing again of COUNT.UP

```
TO COUNT.UP :X
IF :X= 5 [STOP]
```

```
PRINT :X
COUNT.UP :X+1
END
```



The top box represents the "top level" or "level 1" of the procedure, COUNT.UP. It has one input, the number "1." Top level COUNT.UP calls itself, which is the second box, with an input of "2." This is called the second level of the procedure, COUNT.UP. The process continues with recursive calls to the third level, fourth level, and finally fifth level. The fifth level stops the recursive calls. After the fifth level COUNT.UP STOPS, control is passed back to the fourth level COUNT.UP which ENDs, passing control back to the third level which ENDs, passing control back to the second, then to the top level which ENDs. Each subprocedure called, whether it is in a HOUSE or SAILBOAT or a recursive program must run until it ENDs or STOPs. It may be helpful for you to think of the process as occurring in discrete steps over time, as the arrow on the left indicates.

6.1.5 Summary of Simple Recursion

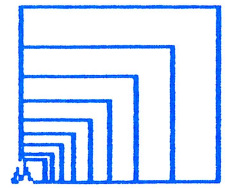
We've seen, in this section, four examples of recursive procedures. By recursive, we mean procedures that call themselves. We write recursive pro-

cedures by including the procedure name in the body of the definition of the procedure. For example, we used TALKALOT in defining TALKALOT, RIGHTBOX in defining RIGHTBOX, SPIRAL in defining SPIRAL, and COUNT.UP in defining COUNT.UP. The effect we've seen thus far is that the recursive call directs the procedure to be repeated. In fact, we need to use the conditional IF statement and STOP to keep the recursion from forever repeating itself.

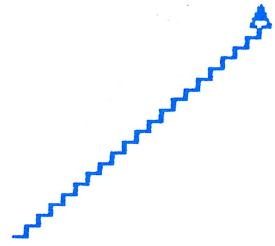
This use of recursion to "loop back" or repeat a procedure at the end of a sequence is called simple recursion or tail-end recursion. Most of the programs in this book use recursion in this way; though we refer to it as "simple," the work it does for us in counting, sorting, listing, drawing, etc. can be quite sophisticated.

Here are some short tail-end recursive programs that create interesting designs. You might try to modify these or add some of your own:

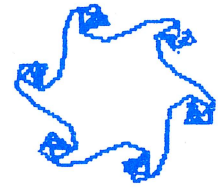
```
?TO GROWBOX :SIZE
>IF :SIZE > 100 [STOP]
>SQUARE :SIZE
>GROWBOX :SIZE*1.3
>END

?GROWBOX 10
```



```
?TO STAIRS :X :Y
>IF :X > 100 [STOP]
>SETX :X
>SETY :Y
>STAIRS :X+5 :Y+5
>END

?STAIRS 5 5
```



```
?TO SPIRE :SIZE :ANGLE :INC
>FD :SIZE
>RT :ANGLE
>SPIRE :SIZE :ANGLE + :INC :INC
>END

?SPIRE 10 20 30
```

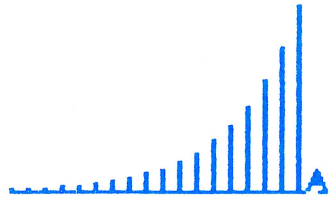


```
?TO GROWTH :SIZE
>IF :SIZE > 100 [STOP]
>FD :SIZE BACK :SIZE
>RT 90 FD 7 LT 90
>GROWTH :SIZE*1.3
>END
```

Note that the recursive call (GROWTH :SIZE*1.3) directs Logo to multiply the value of :SIZE by 1.3 each time the procedure is called. (Top of page 161.)

```
?GROWTH 1
```

Now that we have our programming feet wet with simple recursion, let's jump in all the way and examine "complex recursion."



6.2 Recursion Looks Easy, It Isn't: Complex Recursion

Here is a challenge for you. Examine the program listed below (we call it COMPLEX for good reason) and see if you can predict what will happen when you run it:

```
?TO COMPLEX :X
>IF :X=5 [STOP]
>COMPLEX :X+1
>PRINT :X
>END
```

At first glance, you might be tempted to predict that this recursive program will print a sequence of increasing integers, from 1 to 4. As it turns out, that is not what happens. Type the program in and run it with an input of 1. The following will be displayed on your screen:

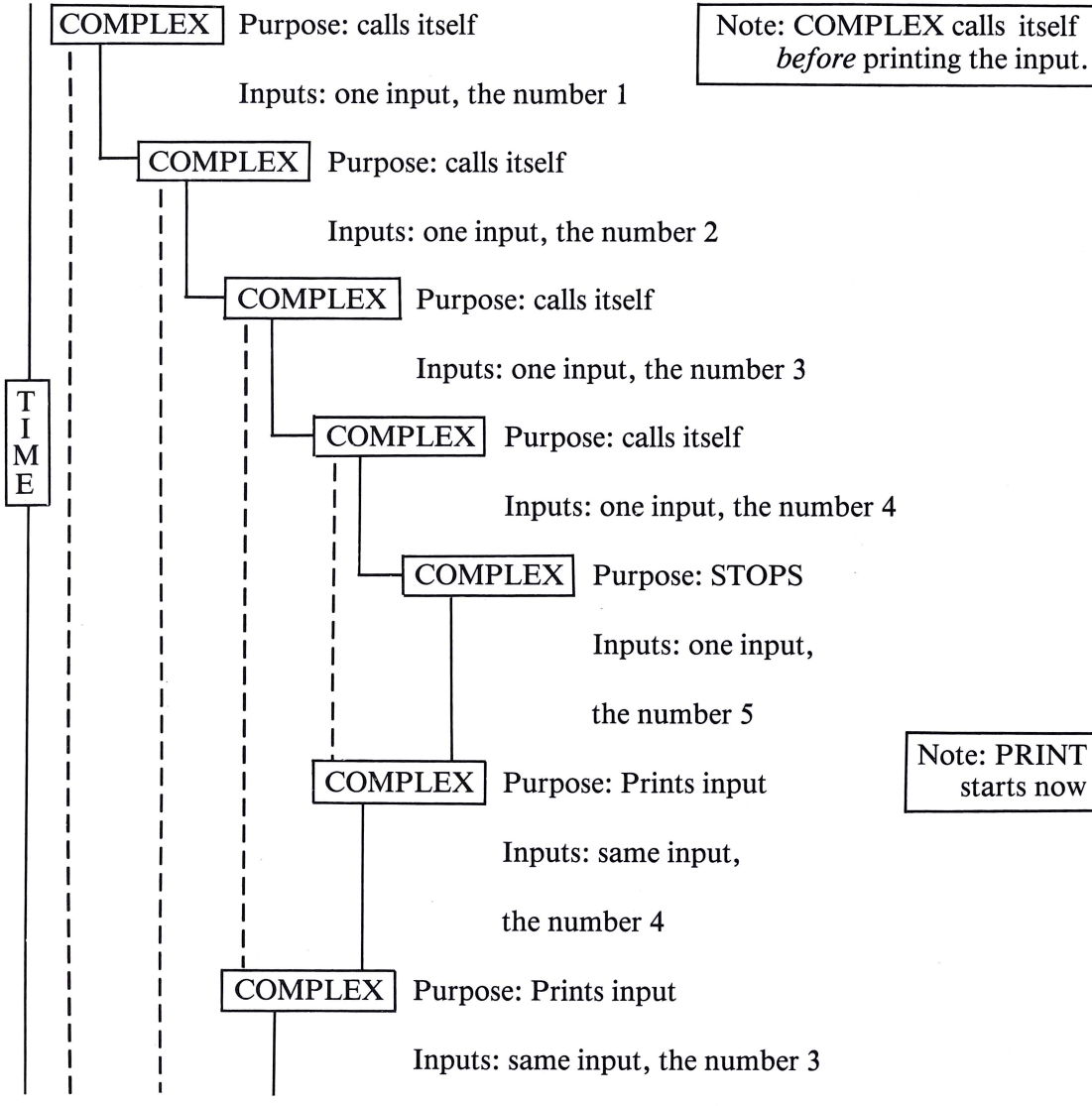
```
?COMPLEX 1
4
3
2
1
?
```

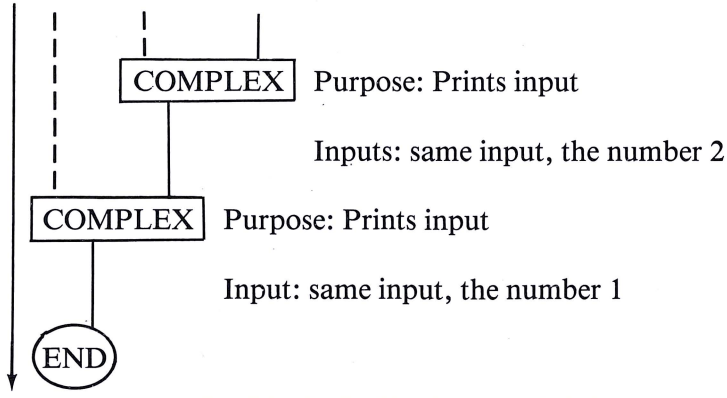
Amazing! How do we explain the display of decreasing integers when the input, :X+1 implies recursive calls with increasing inputs? To begin our explanation, it might be instructive to compare our earlier COUNT.UP program with COMPLEX:

```
?TO COUNT.UP :X          ?TO COMPLEX :X
>IF :X=5 [STOP]          >IF :X=5 [STOP]
>PRINT :X                >COMPLEX :X+1
>COUNT.UP :X+1           PRINT :X
>END                     >END
```

The difference between the two programs occurs in the position or location of the recursive call. In COUNT.UP, the recursive call is at the end (specifically, after the command to PRINT :X). In COMPLEX, it occurs earlier (specifically, before the command to PRINT :X). Why should such a subtle difference account for such an impact on the program execution?

To help explain this, let's diagram the program using box charts:


COMPLEX Purpose: calls itself
Note: COMPLEX calls itself *before* printing the input.
Inputs: one input, the number 1
COMPLEX Purpose: calls itself
Inputs: one input, the number 2
COMPLEX Purpose: calls itself
Inputs: one input, the number 3
COMPLEX Purpose: calls itself
Inputs: one input, the number 4
COMPLEX Purpose: STOPS
Inputs: one input,
the number 5
Note: PRINT starts now
COMPLEX Purpose: Prints input
Inputs: same input,
the number 4
COMPLEX Purpose: Prints input
Inputs: same input, the number 3
TIME



Unlike COUNT.UP, COMPLEX does not print its input until it has finished "COMPLEXing." It continues to call itself and add one to the previous input, until it reaches the number five. We say that COMPLEX 1 created COMPLEX 2, which created COMPLEX 3, etc. until COMPLEX 5 was reached. After COMPLEX 5 STOPs, program control is passed back to COMPLEX 4, which completes each step in its program; it prints its input, 4, then ENDs, passing control back to COMPLEX 3, which now completes each step in its program: it prints its input, 3, then ENDs, passing control back to COMPLEX 2, and so on. Again, the time arrow on the left hand side may be helpful to you in understanding the process.

The dashed lines connecting the two COMPLEX boxes indicate that both boxes are part of the same procedure. However, in between the program lines, a new procedure was called and executed. This is shown below, as we examine what occurs when, say, COMPLEX 3 is called:

```
TO COMPLEX 3
IF :3=5 [STOP]
( . . . COMPLEX 4, COMPLEX 5)
PRINT 3
END
```

We see that the program flow in COMPLEX 3 gets interrupted (before PRINT) to execute COMPLEX 4 (which in turn calls COMPLEX 5).

What we see in COMPLEX is the importance of the location of the recursive call. By moving the recursive call from the end of the procedure to the middle, the effect is different from a simple repetitive action. We call COMPLEX an example of complex recursion. The sequence of events might seem a little vague at first glance; in the next section we'll look at several analogies to help our understanding.

6.3 Recursion Analogies

In this section, we'll examine several examples and analogies to help describe recursion. The metaphors, which vary from legislative procedures to Tom Sawyer painting fences may initially seem quite disparate. They have a common denominator, though—they can be explained (and generated by!) recursive programs.

6.3.1 Who's on First: Keeping Track of Legislator's "Turns"

In governing structures like the United States Senate, there are procedures which allow recognized speakers to "yield" their time to others (usually of similar persuasion) to have the floor. A legislator who obtains permission to speak by a yielding colleague can in turn yield the floor to a third member. This yielding can continue indefinitely (or at least until the number of friends and/or time has run out). Imagine a hot debate occurring in such a legislative setting. Senator 1 has just been recognized; she yields her time to Senator 2, who in turn yields his time to Senator 3, and so on. We'll limit our "and so on" to a maximum of five.

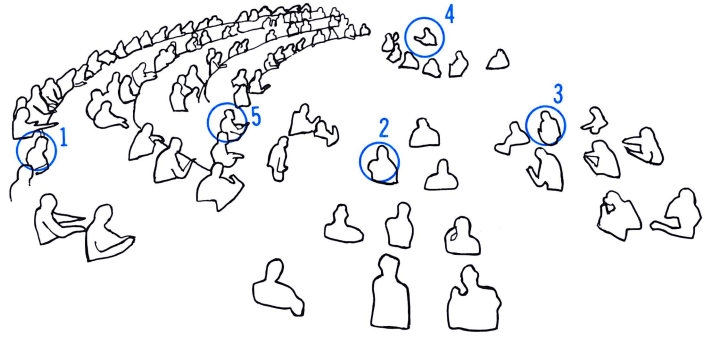


Fig. 6-2. Senatorial chambers.

A recursive procedure that would keep track of "who calls on who" would be as follows:

```
?TO SENATOR.NO :X
>IF :X=5 [STOP]
>PRINT :X
>SENATOR.NO :X+1
>END
```

The name, SENATOR.NO, is an abbreviation for "Senator Number." This simple recursive procedure would display the order in which the senators were called on, i.e. 1, 2, 3, and 4, just like COUNT.UP. If we were legislative watch dogs, and wanted to see the sequence of who has the floor, this simple recursive program would do it.

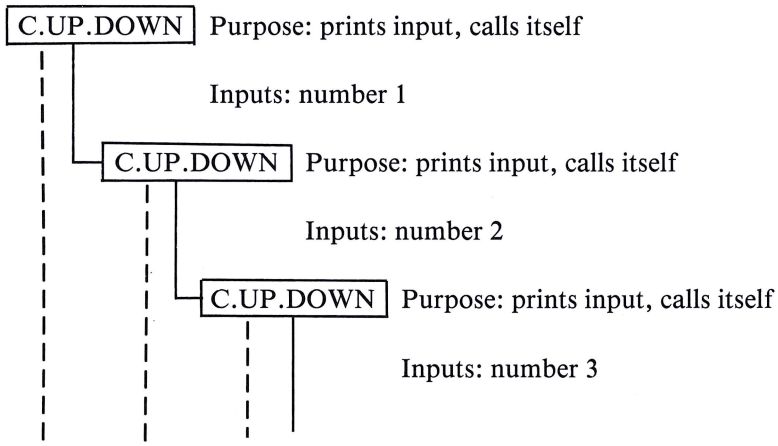
However, and this is the crux of the analogy, senatorial etiquette requires that when Senator number 5 finishes his speech, he thanks Senator number 4 for calling on him, and gives the floor back to Senator number 4. Senator number 4, in turn, thanks Senator number 3 for calling on her, then ENDs her use of the time, passing the floor back to Senator number 3, who ENDs his use of the floor, and so on.

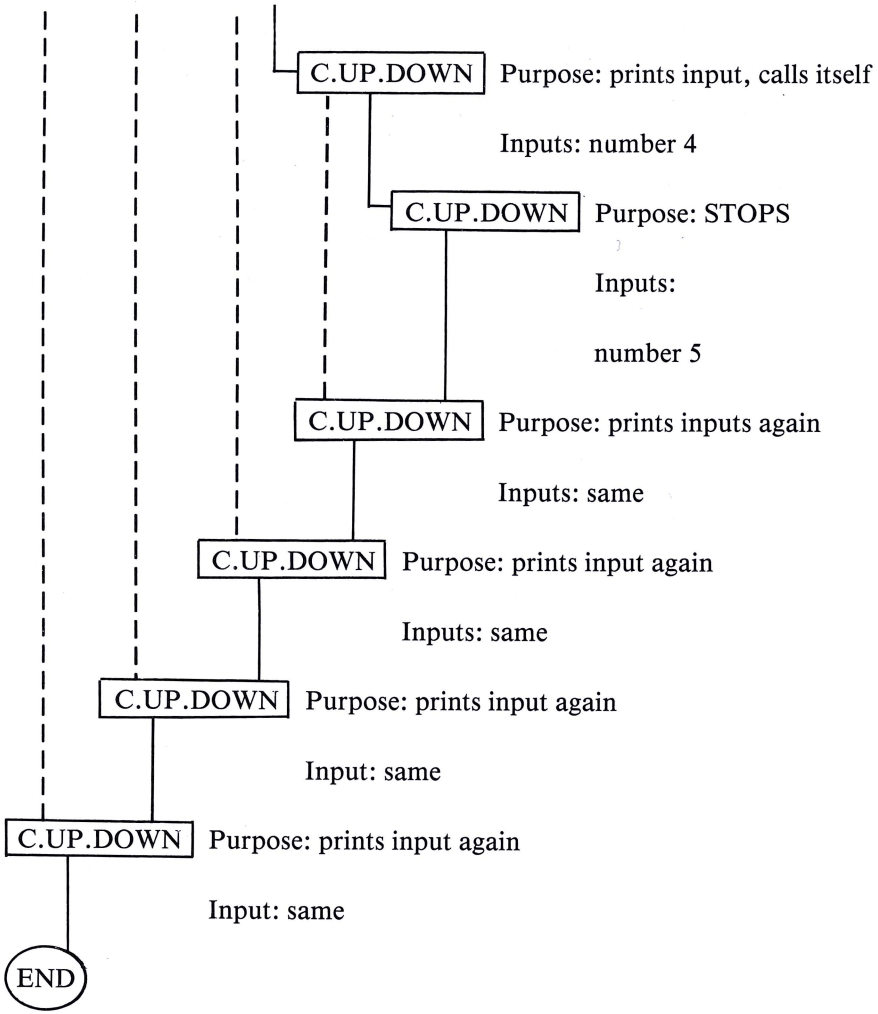
A recursive procedure to keep track of the senators' speaking order, then, would look the procedure COMPLEX. When Senator number five is called on, the senatorial process doesn't end. As happens in COMPLEX, control moves back to 4, 3, 2, and 1. Every procedure called must END or STOP or OUTPUT as discussed in Chapter 5. This means that when :X = 5 STOPs Senator.no 5, it does not END or STOP Senator.no 4 or 3, or 2, or 1. (Or the end of a filibuster?)

It is possible to combine simple and complex recursion into one program. Let's call that program C.UP.DOWN (for count up down):

```
?TO C.UP.DOWN :X
>IF :X=5 [STOP]
>PRINT :X
>C.UP.DOWN :X + 1
>PRINT :X
>END
```

The box diagram for this procedure would be as follows:





This procedure, when executed with an input of 1 will display the following:

```
?C.UP.DOWN 1
1
2
3
4
4
3
```

```
2
1
?
```

Each procedure that is called prints its input twice—once before making the recursive call and once after making the recursive call.

Now let's move into some good 'ole American entrepreneurship for our second recursion metaphor.

6.3.2 Painting Fences

Our second recursion analogy involves a Tom Sawyer-like character who is whitewashing a fence around House #1 on a block with four other such houses with fences.

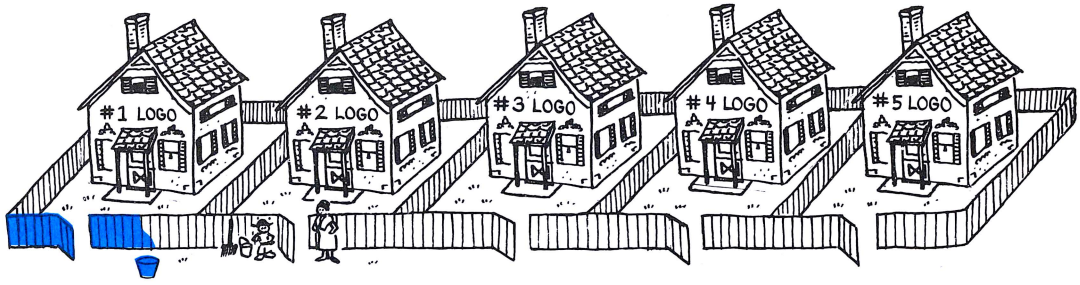


Fig. 6-3. Tom whitewashes a fence.

Halfway through his work on House #1, the neighbor in House #2 comes over and the following dialogue takes place:

Neighbor from House 2: "My, what a great job you're doing painting this fence!"

Tom: "Aw, shucks, ma'am . . . Missus Thackery, from #1 Logo Street here, has offered to pay me seven bucks for doing this here chore for her."

Neighbor: "Seven dollars isn't much for such a hard working boy! My fence could also use a fresh coat of paint. And I'll pay you ten dollars if you start on my fence right away."

Tom: "Ten dollars! It's a deal. Here, can you help me with the brushes and paint?"

(Exit neighbor and Tom, hauling brushes and paint to House #2, Logo Street.)

Halfway through painting house #2, the neighbor from house #3 arrives, offers him even more money for her fence if he would start right away. Tom moves again. This continues until the end of the block, #5 Logo

Street. Owner #5 offers Tom $3.00 per hour. Tom replies "No way!" to owner #5, and continues his work on #4 Logo Street. When he's finished, he moves back to #3, then #2, then #1.

As with COUNT.UP, Tom's moving from houses #1 through #5 could be seen as simple recursion. The main point, however, is in the completion sequence. Tom goes back and finishes each job, but in reverse order.

6.3.3 A Visual Analogy

We interrupt this verbal description of recursion with a visual representation.

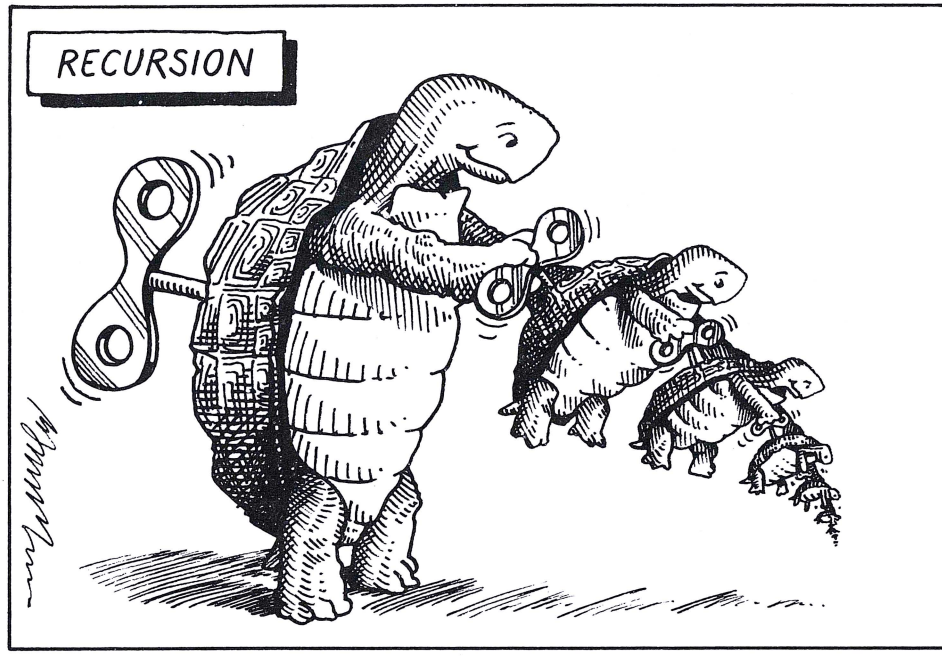


Fig. 6-4. How recursion works.

6.3.4 Recursive Review

So far, we've looked at recursive procedures at three levels of increasing elaborateness. We've used simple or tail-end recursion in procedures without inputs, like TALKALOT or our first RIGHTBOX. We've seen that incorporating inputs into simple recursive procedures, as we did in SPIRAL :DISTANCE :ANGLE and COUNT.UP :X allows us increased flexibility

and control. Finally, in procedures like COMPLEX :X we've seen complex, as opposed to tail-end, recursion at work.

Most of the programs and procedures in this book that use recursion depend on simple, as opposed to complex recursion. We'll conclude this chapter by looking at two more examples.

6.4 Earning Interest

Let's assume we have $1000 in the bank earning interest. How much money will we have after two years? Three years? At first glance, it might not seem necessary to use recursion to calculate interest. After all, we simply multiply the money banked (principal) times the rate of interest to obtain the yearly dividend. But let's say the principal and the interest remain in the bank; the interest earned will also generate interest. Aha! A recursive process, since the interest generated must be added to (i.e. become an input for) the principal that earns interest and so on each dividend year. Suppose the current rate of interest is 7% per year. A procedure to calculate the accrued amount of money (new principal) would involve multiplying the total amount of money by 1.07. What is changing each year is the principal. So let's make the principal an input to a procedure called INTEREST as shown below. We'll stop our program when we exceed $2000.00.

```
?TO INTEREST :PRINCIPAL
>IF :PRINCIPAL>2000 [STOP]
>PRINT :PRINCIPAL
>INTEREST :PRINCIPAL*1.07
>END
```

If we run the program with an initial input of 1000, the display will be as follows:

```
?INTEREST 1000
1000
1070
1144.9
1225.04
1310.8
1402.55
1500.73
1605.78
```

```
1718.19
1838.46
1967.15
2104.85
?
```

In approximately ten years, then, $1000 at 7% interest will double to about $2000.

The procedure INTEREST works because the recursive call, INTEREST :PRINCIPAL * 1.07 reinvests our interest, i.e. it adds it to our original principal to create a new principal for the recursive call.

Can you modify the procedure so that we can vary not only the amount of the principal, but also the interest rate? How could you further modify the procedure so that instead of ending when the principal was greater than 2000, it stops at the end of a given number of years?

Now let's see one convenient way to stop recursive procedures: using a counter.

6.5 Stopping Recursion with an Input

In Section 6.1.2, we created a series of recursive graphics procedures which displayed intricate patterns on the screen. You may have found that if they were not stopped, the screen soon became a tangled mass of turtle lines.

How can we control such procedures so that we can prevent the clutter? One way, as we saw, was the emergency CTRL-G sequence. This command stops the procedure at a random point in the execution. A second way of limiting graphics procedures would be to use the FENCE command discussed in Chapter 2. FENCE puts limits on the turtle's movements by not allowing the lines to WRAP around. A third way of controlling the procedure would be to use an input to limit the number of recursive calls.

We'll call this input "counter." As we discussed earlier, we defined the first call of a procedure "top level" or "level 1." The next repetition of the procedure is level 2, followed by level 3, and so on. In essence, we want the recursive procedure to call a given number of levels; we'll use COUNTER to keep track of the levels. We'll modify SPIRAL to include this variable:

```
?TO SPIRAL :DISTANCE :ANGLE :COUNTER
>IF :COUNTER=0[STOP]
```

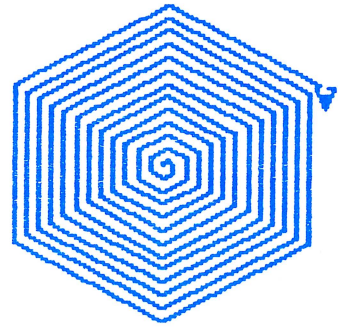
```
>FD :DISTANCE
>RT :ANGLE
>SPIRAL :DISTANCE + 1 :ANGLE :COUNTER -1
>END
```

Notice how we use the recursive call to decrease the COUNTER input by one each time we call SPIRAL. Eventually, the COUNTER input reaches zero and that level stops. (Note: our COUNTER variable does not correspond to the recursive "level.")

Clear the screen and eliminate the bottom lines of text with FULLSCREEN as we did in Chapter 2. Now call

```
?SPIRAL 1 60 75
```

and you'll see the following displayed:



Here are two more nice designs.

```
?SPIRAL 80 175 38
```

and

```
?SPIRAL 5 88 100
```

The procedures are now repeating as many times as we want, and then stopping. Adding a variable such as :COUNTER is a powerful way to control the amount of recursion we want in a procedure.

6.6 Summary

Here are some of the important points in this chapter:

1. Recursive procedures are those which call themselves. At the simplest level, recursive procedures are repetitive; they appear to loop back to repeat commands.
2. Recursive procedures create new procedures at lower levels. When a lower level STOPs or ENDs, control is passed back up to the next higher level, eventually reaching the top level which is the Command Level.
3. Complex recursion is not simple repetition; the position of the recursive call within the body of a procedure can produce dramatic changes in a program's execution.
4. Recursive procedures can be stopped with the keyboard sequence CTRL-G, or by including an IF–condition–STOP statement in the body of the procedure.
5. Recursive procedures and programs can be analyzed through the box chart approach to understand their sequence of execution.

6.7 Review Questions and Exercises

Review Questions

1. Give one example where recursion and REPEAT produce the same effect.
2. How do you give an emergency stop to halt runaway recursion?
3. In the following program

```
TO COUNTA :X
PRINT :X
COUNTA :X+1
END
```

   a. When does the program end?
   b. Can you use a REPEAT command to do the same thing?

4. If a recursive program STOPs naturally, using an IF–condition–STOP sequence at the 4th level, do the 3rd, 2nd, and 1st levels STOP or END?
5. In a recursive procedure, what level procedure always ends last?

6. In the following procedure, run with the input shown,

```
TO COUNTB :X
IF :X>10 [STOP]
COUNTB :X + 1
PR :X
END
?COUNTB 5
```

   a. What number is printed first?
   b. What number is printed last?
   c. What is the highest number printed?

Exercises at the Terminal

1. Try drawing 24 boxes, each one rotated 15 degrees from the previous one
   a. using REPEAT
   b. using recursion
2. Draw 12 boxes using recursion with each one drawn bigger than the previous one.
3. Draw 24 boxes with each one rotated by 15 degrees and bigger by 15% than the previously drawn box.
4. a. Draw spirals with increasing angles not sizes
   b. Draw spirals with both angles and sizes increasing
5. Complex Recursion! Here are two simple graphics programs that produce amazing results with various inputs.

```
TO NEST.TRI :SIZE
IF :SIZE < 10 [STOP]
REPEAT 3 [NEST.TRI :SIZE/2 FD :SIZE RT 120]
END

TO NEST.POLY :SIZE :N :DEPTH
IF :SIZE < 10 [STOP]
REPEAT :N [NEST.POLY :SIZE/:DEPTH :N :DEPTH FD :SIZE
RT 360/:N
END
```

Answers to Review Questions

(1) These two programs would produce the same results:

```
TO FIVE.NAME
REPEAT 5 [PRINT "MARIJANE]
END
```

```
TO FIVE.NAME :X
IF :X>5 [STOP]
PRINT "MARIJANE
FIVE.NAME :X+1
END
```

FIVE.NAME FIVE.NAME 1

(2) CTRL-G (3) a. When the largest number possible in Logo, 2147483647, is reached. b. No (4) END (5) The first level (top level) procedure always ends last. (6) 10, 5, 10

CHAPTER 7

NUMBERS, WORDS AND LISTS

CONCEPTS:

Logo data objects
identifying numbers, words and lists
taking apart numbers, words and lists
when lists are not lists

PRIMITIVES:

FIRST, BUTFIRST, LAST, BUTLAST

7.0 Introduction

In the first six chapters we have used the turtle to lay the foundation for learning Logo. We have learned Logo's underlying structure of primitives and inputs and we have learned how to plan and write programs. It is now time to add the framework to the foundation, put meat on the bones, to learn the "rest of Logo." Chapters 7 and 8 will discuss the three basic inputs: numbers, words and lists and Chapters 9 and 10 will bring us variables and Logo's math handling capabilities. We will start with numbers, words and lists.

We can think of a computer program as an information factory. Like all factories, the information factory requires raw material for input. In Logo, the raw materials are called "data objects," of which there are three kinds. Logo's data objects are numbers, words and lists. This information factory

crunches, munches, molds and processes (data processing) the data, eventually yielding the finished product to our great satisfaction.

Our job is to program the factory; create the instructions for processing the data. We want to make our information factory, APPLE LOGO, work smoothly and efficiently. To do this we need to start by knowing three important things about the Logo factory's raw materials or data objects.

1. How do we *clearly identify* numbers, words and lists?
   The factory needs to have labels or tags on all incoming materials in order to correctly process them.
2. How do we take numbers, words and lists *apart?*
   Fortunately, we have such well-oiled machines as the Logo primitives, FIRST and BUTFIRST, which are specially designed to sort and separate incoming data.
3. How do we put numbers, words and lists *together?*
   More tools or primitives such as WORD and SENTENCE put humpty-dumpty back together again.

In this chapter, we will answer the first two questions. The third question will be the topic of chapter 8.

7.1 How Do We Identify Numbers, Words and Lists

A Logo primitive or procedure needs a way of determining whether you're giving it a number, a word or a list as input. Logo makes this determination by reading the first character fed to the primitive or procedure. The method for specifying each data object is fairly simple:

a. Numbers look like numbers.

```
707 3.14 1984 -4.6
```

b. Words usually begin with a quote and end with a blank space.

```
"HI "CALIFORNIA "X1 "WHO?
```

c. Lists always begin and end with a bracket.

```
[HI] [SAN FRANCISCO, CA] [10 20 30]
```

We will discuss the important topic of numbers and their arithmetic operations in detail in Chapter 10. Now let's take a look at words and lists. Words and lists are Logo's equivalent to words and sentences in the English language. They are absolutely essential to our writing procedures that use language, store addresses and phone numbers, or graph data, etc.

7.1.1 Words

A Logo word is made up of a "string of characters" beginning with a quote mark (") and usually ending with a space. We use the word "string" because each character is followed by another character and the sequence of characters is strung together, uninterrupted by a space. A Logo word is not the same as a word in the dictionary: it does not have to make sense or be limited to letters. Here are some examples:

Logo words

```
"PRINCESS
"WHAT?
"HAIR.RAISING
"XP303YY
"U&IR0K
```

The following two examples would not be interpreted as one Logo word.

```
"SAN FRANCISCO (1 word and 1 unidentified object FRANCISCO)
"HAIR-RAISING  (uses dash sign "—", see explanation below)
```

The last example is incorrect because it uses a Logo "special character," in this case, the dash sign (—). The dash sign (—), the plus sign (+), along with the left bracket ([), right bracket (]), parentheses ( ), equal sign, (=), and asterisk, (*), all have the same effect on words as a blank space. That is, they end the word. These special Logo characters are known as "delimiters," since they signal the end or the limit of the word. They "delimit" the word. Thus, in the example, "HAIR—RAISING consists of one word HAIR ended by the dash sign (—) and one unidentified object, RAISING.

If you want to include any of these special characters (delimiters) in a word as normal alphabetical characters, you can place a backslash, (\), (typed with a CTRL-Q on the Apple), before the character. This process of treating special characters normally by using a backslash is called using an "escape sequence." Here are two examples with an escape sequence.

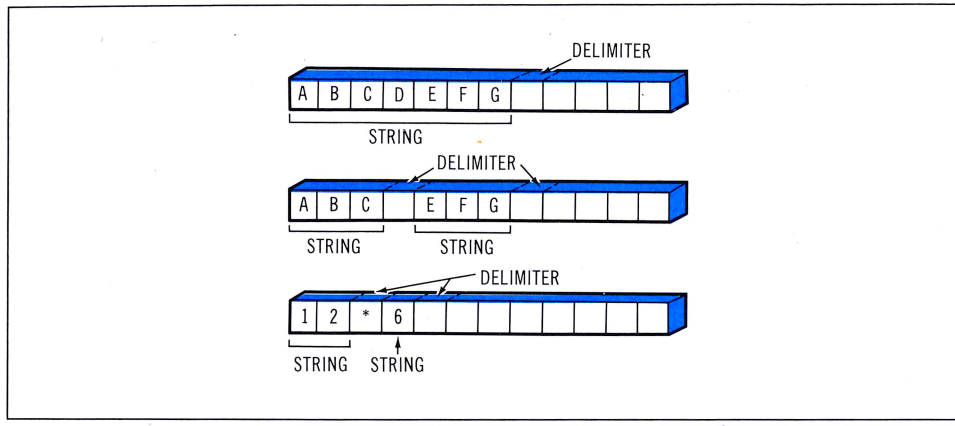


Fig. 7-1. Character strings.

Using an escape sequence

```
"JACK" +JILL
```
(produces one word, JACK+JILL)

```
"7 *5
```
(produces one word, 7*5)

Since words must end with a space or special character (delimiter), a period, ".", (which is not a special character in Logo), is sometimes used to piece together two or more English words as one Logo word. Thus, we can write

```
"JACK.JILL
"HAVE.A.NICE.DAY
"WHAT.IS.4 +7?
"ONE.LOGO.WORD
```

The period helps make the one Logo word more readable to us. To Logo, it is still just one string of characters.

Remember that Logo words are always used as inputs to primitives and procedures like PRINT.

```
?PRINT "AMERICAN
AMERICAN
?PRINT "A.WORD
A.WORD
```

7.1.2 Unmarked Words

What happens if we give Logo an unmarked word, such as

```
HI
```

or

```
"SAN FRANCISCO
```

If each case of unmarked words, Logo checks to see if it is a primitive or if you have defined a procedure with the name of the word. If it does not find a primitive or a procedure by that name, Logo gives the familiar error message

```
I DON'T KNOW HOW TO HI
```

or

```
I DON'T KNOW HOW TO FRANCISCO
```

We can summarize Logo words with the following picture.

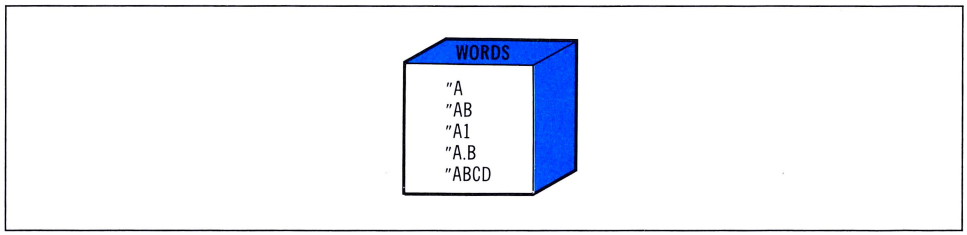


Fig. 7-2. Logo words.

7.1.3 Lists

A Logo list is made up of Logo data objects and is always enclosed in brackets. We can have a list of numbers, a list of words, a list of lists or even a list of some combination of numbers, words and lists. For example, we can have lists that contain a question or answer, inventories or monthly sales records. Here are some examples.

Logo lists

```
[HI]
```

```
[WHAT IS THE WORLD'S FINEST CITY?]
[[SAN FRANCISCO] CALIFORNIA]
[1972 1976 1980 1984]


[ ]
```

Whereas words are made up of characters in a string, lists are made up of Logo data objects that we also call "elements." Each of the following lists contains 7 elements, since each data object in the list is one element.

```
[1 2 3 4 5 6 7]
[THIS IS A LIST OF 7 ELEMENTS]
[ONE SMALL STEP FOR MAN . . . [ARMSTRONG]]
```

In the last example the three dots " . . . " (a character string) are one element as is the list [ARMSTRONG].

Notice that words contained in a list are not quoted. Each element in the list is delimited by spaces or by the delimiting bracket characters, "[" or "]".

Lists, like words, are always used as input (raw material) to primitives and procedures. In each example below, we give PRINT some lists as input:

```
?PRINT [5*6]
5*6
?PRINT [THE BEST THINGS IN LIFE ARE FREE]
THE BEST THINGS IN LIFE ARE FREE
?PRINT [10 20 30 40 [150 160]]
10 20 30 40 [150 160]
```

Notice that the outside brackets are removed when PRINT prints a list. If we print a list with just a single word, it appears no different than printing a word.

```
?PRINT [HELLO]
HELLO
?PRINT "HELLO
HELLO
```

However, a list is *not* equal to a word, so even though the PRINT command produces the same result, the statement

```
[HELLO] = "HELLO
```

is false.

A very useful list is the empty list, [ ]. This list is often added to procedures to produce a blank line that improves the readability of programs as shown in the quotation below:

```
TO LOOKNICE
PR [ ]
PR [IF YOU THINK EDUCATION IS EXPENSIVE,]
PR [ ]
PR [TRY IGNORANCE]
PR [ ]
END
```

Run this procedure by typing its name

```
?LOOKNICE
IF YOU THINK EDUCATION IS EXPENSIVE,
TRY IGNORANCE
?
```

You might think that it would look better to center the second line underneath the first line, by writing the second line as

```
PR [  TRY IGNORANCE]
```

This will not work, since Logo removes blank spaces when printing lists (except for the one space between words). There are ways to add spaces when printing text. We will show them to you later in this chapter.

The receiving department at the factory now knows how to tell whether the incoming data objects are numbers, words or lists. Let's step inside the factory and see what we can do with these objects.

WHEN IS A LIST NOT A LIST

Lists were created to hold data objects. The elements of a list must be numbers, words and lists. When we take lists apart, we do *not* get dotted words, primitives or procedures. Unmarked words in lists are considered to be words and they are considered "quoted" if taken out of a list.

There are three exceptions to this rule. When lists are used as input to the primitives,

```
REPEAT RUN IF
```

they are not lists of data objects. Lists used with these three primitives must contain primitives or procedures that are to be run or executed. For example,

```
REPEAT 4 [FD 40 RT 90]
```

These lists associated with these three primitives are not real lists. (The primitive DEFINE takes a list as input. Each list must be a command or input. See Appendix A)

7.2 Taking Words and Lists Apart

What can Logo do with numbers, words and lists? In fact, what can any computer do with its data objects? The answer is: a computer can inspect, analyze, compare, separate, store, print, or discard its data objects. In other words, Logo provides tools for isolating and combining characters in a word and elements in a list. Logo provides us with four primitives, all operations, for taking words and lists apart. These operations are FIRST, BUTFIRST, LAST, and BUTLAST. Each of these primitives selects and outputs part of a word or list. For example,

```
FIRST "LOGO
  outputs L


LAST "LOGO
  outputs O
```

Although FIRST and LAST are defined similarly, we will discuss FIRST and BUTFIRST together, since they team up to nicely dissect words and lists as we shall show later.

7.2.1 The Primitives FIRST and BUTFIRST

The primitive FIRST is used to select the first character from a word or the first element from a list.

```
FIRST "ABRAHAM
  outputs A


FIRST [ABRAHAM LINCOLN]
  outputs ABRAHAM
```

Here is our formal definition of FIRST.

FIRST	**(takes one input)**
Purpose	Selects and outputs the first element from the input
Abbrev	None
Inputs	Any Logo data object; number, word or list
Outputs	If the input is a number or word, FIRST outputs the first character If the input is a list, FIRST outputs the first element of the list.

Examples:

```
?FIRST "HELLO
I DON'T KNOW WHAT TO DO WITH "H
```

Remember! Since FIRST is an operation, its output must be used as input to a command or another operation. What we should have typed is

```
?PRINT FIRST "HELLO
H


?PRINT FIRST [KENNEDY JOHNSON NIXON CARTER REAGAN]
KENNEDY


?PRINT FIRST 1984
1
```

Here is a different example where the input list contains other lists:

```
?PRINT FIRST [[GREG MARTIN] [SAN ANSELMO] [415-2001]]
GREG MARTIN
```

Notice that FIRST just operates on the outer list.

What happens if we give FIRST an empty word (") as input?

```
?PRINT FIRST "
FIRST DOESN'T LIKE " AS INPUT
```

What comes after the " sign is a space, so " is the empty word.

```
?PRINT FIRST [ ]
FIRST DOESN'T LIKE [ ] AS INPUT
```

Although FIRST does not like the empty word or the empty list as input, the empty word and empty list are valid data objects. They can be used to provide space between lines and they are often used to stop runaway recursive programs as we shall see later in this section.

We can string the FIRST operators together. What would the following commands give?

```
?PRINT FIRST FIRST [NEW YORK]
```

If you answered "N," that's good! Here is a trickier question. What would the following produce?

```
?PRINT FIRST FIRST FIRST [NEW YORK]
```

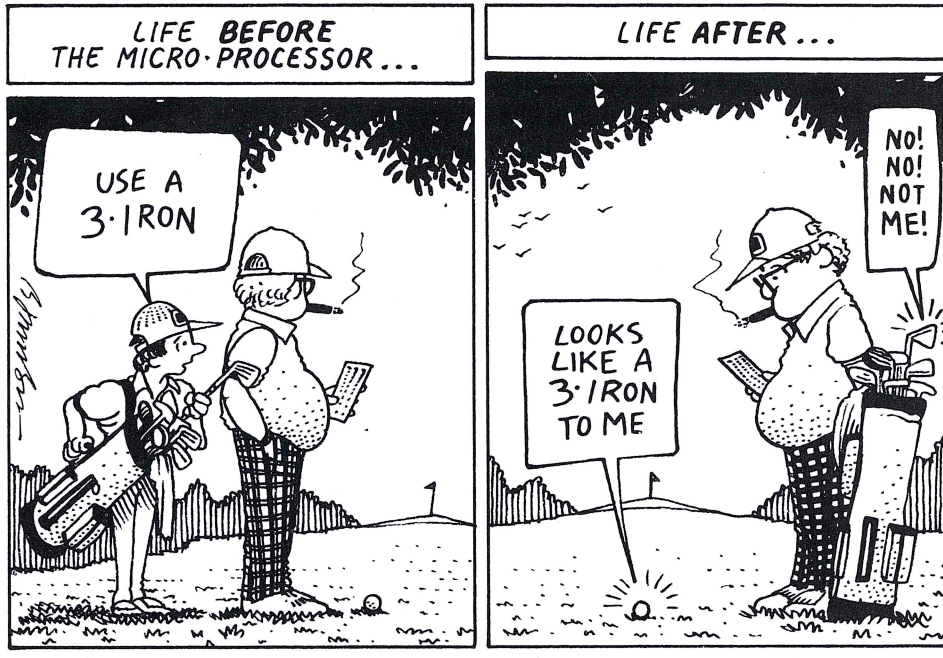
Again it prints an "N. As you might guess, if we add more FIRSTs, each one would output the letter N to be fed to the preceding FIRST.

Here is a procedure that takes a list as input and checks the first element of the input to see if it is the number 7.

```
TO CHECK :NUMLIST
IF (FIRST :NUMLIST) = 7 [PRINT "WINNER STOP]
PRINT "LOSER
END


?CHECK [4 5 6 7 8]
LOSER
```

We use parentheses around FIRST :NUMLIST to make sure that Logo compares the first number in the list and not the whole list. Without the parentheses, Logo would compare :NUMLIST with 7, output the word "FALSE, then take FIRST of the result, which is not what we want to do. These finer points on using the IF statement are discussed fully in Chapter 11.

The procedure above doesn't do much because it can only check the first number on the input list. What we often want is to check the whole list for



a specific number. The best way to do that is to use FIRST and BUTFIRST together with recursion. Let's look at the primitive BUTFIRST and then come back and rewrite this procedure.

The primitive BUTFIRST is used to take *all but the first* character from a word or element from a list as shown below.

```
BUTFIRST "FLUNKY
  outputs LUNKY

BUTFIRST [SUE LIKES ROSES]
  outputs LIKES ROSES
```

In Fig. 7-3, notice how FIRST and BUTFIRST join forces to completely encircle words and lists.

Here is our formal definition of BUTFIRST.

BUTFIRST	**(takes one input)**
Purpose	Selects and outputs all but the first element from the input

Abbrev	BF
Inputs	Any Logo data object; number, word or list
Outputs	If the input is a number or word, BUTFIRST outputs all but the first character If the input is a list, BUTFIRST outputs all but the first element.

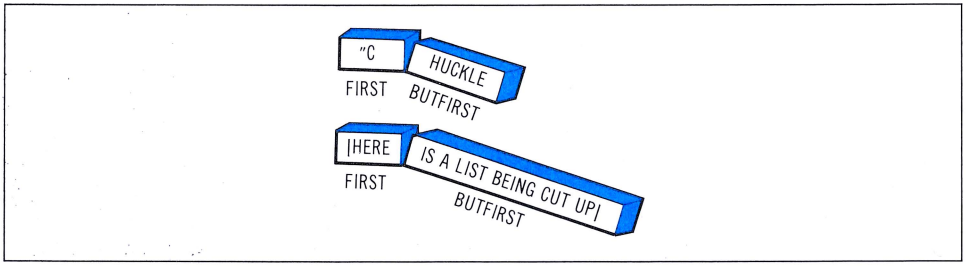


Fig. 7-3. FIRST and BUTFIRST operating on a word and on a list.

Examples:

```
?PRINT BUTFIRST "HELLO
ELLO


?PRINT BUTFIRST [KENNEDY JOHNSON NIXON CARTER REAGAN]
JOHNSON NIXON CARTER REAGAN


?PRINT BUTFIRST 1984
984
```

What happens when we give BUTFIRST a list containing other lists? Just as with FIRST, it only operates on the outer list.

```
?PRINT BUTFIRST [[GREG MARTIN] [SAN ANSELMO] [415-2001]]
[SAN ANSELMO] [415-2001]
```

BUTFIRST, like FIRST, does not like the empty word or the empty list as input.

We can also string the BUTFIRST operators together like this:

```
?PRINT BUTFIRST [NEW YORK]
YORK


?PRINT BUTFIRST BUTFIRST [NEW YORK]

?
```

In the last example, BUTFIRST returns the empty list, [ ], which is a valid data object. However, if we try BUTFIRST one more time

```
?PRINT BUTFIRST BUTFIRST BUTFIRST [NEW YORK]
BUTFIRST DOESN'T LIKE [ ] AS INPUT
```

Even though the empty list, [ ], is a valid data object, using BUTFIRST on the empty list does not make sense and Logo complains with an error message.

7.2.2 Sample Procedures: Cutting Words and Lists

Now that we know how FIRST and BUTFIRST operate on words and lists, let's give them a job to do. The most common way these two partners work together is in a recursive procedure that takes apart words and lists. Here is our procedure:

```
TO CUTWORD :WORD
IF :WORD = '' [STOP]
PRINT :WORD
CUTWORD BUTFIRST :WORD
END


?CUTWORD "CANDY
CANDY
ANDY
NDY
DY
Y
?
```

The most important point to recognize here is that BUTFIRST *changes*

the inputs to each recursive call of the procedure. This is so important that we would like to emphasize the effect of BUTFIRST on the inputs with the following diagram:

```
CUTWORD "CANDY
 CUTWORD "ANDY
  CUTWORD "NDY
   CUTWORD "DY
    CUTWORD "Y
     CUTWORD "  (the empty word as input stops this procedure)
```

What would you expect to happen if we give CUTWORD a list for an input?

```
?
?CUTWORD [2 4 6 8 WHO DO WE APPRECIATE?]
2 4 6 8 WHO DO WE APPRECIATE?
4 6 8 WHO DO WE APPRECIATE?
6 8 WHO DO WE APPRECIATE?
8 WHO DO WE APPRECIATE?
WHO DO WE APPRECIATE?
WE APPRECIATE?
APPRECIATE?
BUTFIRST DOESN'T LIKE [ ] AS INPUT
```

We see that the procedure runs okay until the input is the empty list, [ ]. This particular IF statement is designed to stop the recursive calls once the input is an empty word, (" "), but not the empty list. What can we do about that? Here are three suggestions. First, we could define a new procedure, perhaps calling it CUTLIST, that contains the appropriate IF statement. Another approach would be to add a new IF statement to the CUTWORD procedure like this:

```
TO CUTWORD :WORD
IF :WORD = "  [STOP]
IF :WORD = [ ] [STOP]
PRINT :WORD
CUTWORD BUTFIRST :WORD
END
```

This revised procedure will now take both lists and words as inputs. However, the names we gave to the procedure and its inputs are not as appropriate (the procedure still runs no matter what names we use). A third solution to the problem uses a new primitive, EMPTYP, that we formally introduce in Chapter 11.

```
TO CUTALL :OBJECT
IF EMPTYP: OBJECT [STOP]
PRINT :OBJECT
CUTALL BF :OBJECT
END


?CUTALL [FOR HE'S A JOLLY GOOD FELLO]
FOR HE'S A JOLLY GOOD FELLO
HE'S A JOLLY GOOD FELLO
JOLLY GOOD FELLO
GOOD FELLO
FELLO
```

The last of these procedures has the fewest number of lines and handles both words and lists as inputs. Experienced programmers would call it the most "elegant" of the four procedures.

Let's try one more example. Let's go back to the problem from the last section where we want to test to see if a specific number is in a list. Here is our revised procedure.

```
TO CHECKA :NUMLIST
IF (FIRST :NUMLIST) = 7 [PRINT "WINNER STOP]
CHECKA BF :NUMLIST
END


?CHECKA [4 5 6 7 8]
WINNER
```

The first program line does the checking for the number 7. It does this by taking the first number from the list, in this case 4, and comparing it with the number 7. Since they don't agree, we move to the next line which calls the procedure CHECKA again (recursively) but with a new input, the list [5 6 7 8]. The process of checking for a 7 continues until it finds one and prints WINNER. (Here is a reminder about recursion. The recursive level

that prints WINNER STOPs, passing control back up to the next level, which ENDs, passing control up to the next level, which ENDs, etc.)

To keep this procedure simple, we have omitted instructions for handling a list which does not contain the number 7. If the number 7 is missing from the input list, the procedure runs until FIRST tries to use the empty list for input [ ], which returns the error message,

```
FIRST DOESN'T LIKE [ ] AS INPUT
```

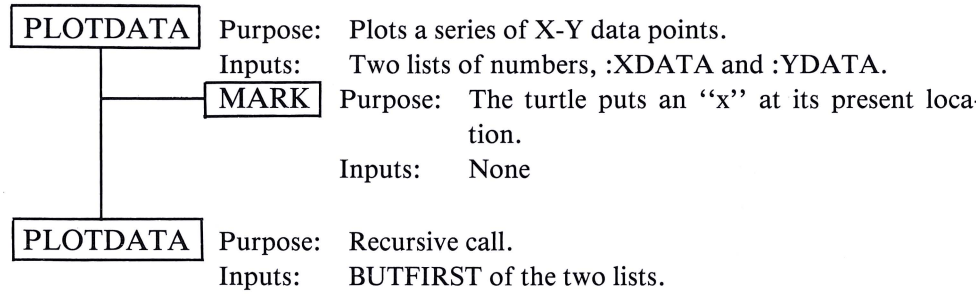
Can you add a line to stop the recursive calls from reaching this level?

7.2.3 Sample Program: Plotting Data

Here is a sample program that gives a nice example using FIRST and BUTFIRST to plot two lists of data. In this example, our two data lists are called XDATA and YDATA and contain the following data points:

XDATA 20 40 60 80 100
YDATA 10 20 30 40 50

Since this procedure contains the subprocedure, MARK, we will first give an overview of the program using the box chart.


PLOTDATA
Purpose: Plots a series of X-Y data points.
Inputs: Two lists of numbers, :XDATA and :YDATA.
MARK
Purpose: The turtle puts an "x" at its present location.
Inputs: None
PLOTDATA
Purpose: Recursive call.
Inputs: BUTFIRST of the two lists.

Here are the two procedures along with a sample run:

```
TO PLOTDATA :XDATA :YDATA
IF :XDATA = [ ] [STOP]
IF :YDATA = [ ] [STOP]
PU
SETX FIRST :XDATA
SETY FIRST :YDATA
MARK
```

```
PLOTDATA BF :XDATA BF :YDATA
END


TO MARK
PD
RT 45
FD 2 BK 4 FD 2
RT 90
FD 2 BK 4 FD 2
LT 135
PU
END


?PLOTDATA [20 40 60 80 100] [10 20 30 40 50]
```

There are two key ideas needed to understand how this program works. The heart of the program uses the primitives SETX and SETY (which we've used before) to move the turtle to the first X-Y data points, X=20 and Y=10, in this case. Then the subprocedure MARK places an "X" on the screen.

The second major point involves using recursion to call PLOTDATA again, however with a reduced set of X-Y data points created with BUTFIRST. Again SETX and SETY move the turtle to the new X-Y data points, X=40 and Y=20. The recursive call to PLOTDATA keeps reducing the inputs :XDATA and :YDATA until it reaches the empty list "[ ]", and stops.

You are probably thinking of various ways to improve this program. For example, you can

1. Add an X-Y axis.
2. Instead of plotting "x"s, make this into a bar graph or a line graph.
3. Find an easier way to enter the data or to change data.
4. Be able to handle a wider range of input numbers in XDATA and YDATA.

The first two ideas are things you could do right now. The last two suggestions take additional Logo tools that we will introduce in later chapters. Chapter 13 contains a full-blown graphing package with lots of advanced features.

To summarize, FIRST and BUTFIRST allow you to easily access the first element or all but the first elements of a list (or characters in a word or numbers in a number). There is an equivalent way to do the job from the rear, using primitives called LAST and BUTLAST (naturally).

7.2.4 The Primitives LAST and BUTLAST

The primitives LAST and BUTLAST are defined and used just like FIRST and BUTFIRST. For example

```
LAST "BALL
  outputs L


LAST [BASEBALL FOOTBALL [PING PONG BALL]]
  outputs PING PONG BALL
```

Here is our formal definition.

LAST	**(takes one input)**
Purpose	Selects and outputs all the last element from the input
Abbrev	None
Inputs	Any Logo data object; number, word or list
Outputs	If the input is a number or word, LAST outputs all the last character If the input is a list, LAST outputs the last element

Examples:

```
?PRINT LAST "JELLO
O


?PRINT LAST 415.1984
4
```

```
?PRINT LAST [LETTUCE RADISHES CARROTS PEACHES]
PEACHES
```

In the last example, LAST kept the veggies and removed the fruit, which seems like a good idea, since its hard on the digestive system to mix fruits and vegetables.

Just as the primitive BUTFIRST complements FIRST, the primitive BUTLAST works hand in glove with the primitive LAST. BUTLAST can be thought of as selecting *all but the last* character in a word or element in a list as shown below.

```
BUTLAST "ORANGE
  outputs ORANG
```

Here is a figure showing LAST and BUTLAST followed by a formal definition of BUTLAST.

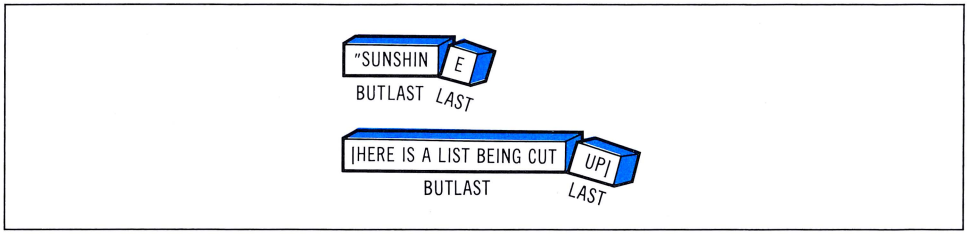


Fig. 7-4. LAST And BUTLAST operating on a word and list.

BUTLAST	**(takes one input)**
Purpose	Selects and outputs all but the last element from the input
Abbrev	BL
Inputs	Any Logo data object; number, word or list
Outputs	If the input is a number or word, BUTLAST outputs all but the last character If the input is a list, BUTLAST outputs all but the last element.

Examples:

```
?PRINT BUTLAST "HAPPY
```

```
HAPP


?PRINT BUTLAST [YANKEES INDIANS [RED SOX] RAMS]
YANKEES INDIANS [RED SOX]
```

Once more we see the power of Logo's list-handling primitives. In this example, BUTLAST correctly removes the RAMS from the list, since they were "playing out of their league," so to speak.

Here is an example procedure that makes words and lists look like triangles by removing the last letter or element.

```
TO GOBBLE :INPUT
IF EMPTYP :INPUT [STOP]
PRINT :INPUT
GOBBLE BUTLAST :INPUT
END


?GOBBLE "MISSISSIPPI
MISSISSIPPI
MISSISSIPP
MISSISSIP
MISSISSI
MISSISS
MISSIS
MISSI
MISS
MIS
MI
M
```

Be sure to convince yourself that this works. Concentrate on what inputs get fed to GOBBLE at each level.

The same trick can be done with a list as input, although the list does not look quite as nice.

7.3 Summary

Here are the important points in this chapter.

1. Numbers, words and lists are the basic "data objects" in Logo.
2. All inputs, variables and operations eventually have a number, word or list as their value.
3. Numbers are identified by, 1.2 −3.4, 2E5.
4. Words are identified by leading quotes, "HELLO.
5. Lists are identified by surrounding brackets [ALPHA BETA GAMMA DELTA]
6. Numbers, words and lists can be taken apart with the operations

```
FIRST  BUTFIRST  LAST  BUTLAST
```

7. Lists used as input to REPEAT, IF and RUN must contain primitives or procedures rather than Logo data objects.

7.4 Review Questions and Exercises

Review Questions

1. What do the following characters have in common?
   +, −, [,], (,), *, =
2. How can you print the statement (4+3)*5 without Logo doing the arithmetic?
3. How does Logo interpret an unmarked word (no quotes and no dots)?
4. What kind of data objects can be placed in lists?
5. What will FIRST "HELLO output?
6. What will FIRST FIRST "HELLO output?
7. What will FIRST BUTFIRST "HELLO output?
8. What will FIRST BUTLAST "HELLO output?
9. Give an example where the FIRST equals the LAST. Do this for a list input and a word input.

Exercises at the Terminal

1. Write a procedure that counts the number of letters in a word and prints the results.

2. Write procedures that are like CHECKA in the text and that do the following:

   a. Finds a word in a list; call the procedure
      FIND.W :WORD :LIST
   b. Finds a character in a word; call the procedure
      FIND.C :CHAR :WORD
   c. Finds a list within a list; call the procedure
      FIND.L :LIST1 :LIST2

   In each case above, when the item is found, print the sentence I FOUND XXXX where XXXX is the item. If no item is found, print the sentence XXXX IS NOT HERE
3. The primitive DOT takes a list containing two numbers, say X and Y as input, and places a dot on the screen at that location.

   Write a procedure that takes two lists of data, called XLIST and YLIST, as input and graphs the data in the lists, using DOT. Call the procedure

```
GRAF :XLIST :YLIST
```

Answers to Review Questions

(1) They end or delimit the word. (2) PRINT [(4+3)*5] (3) As a primitive or procedure. (4) All three kinds (5) "H (6) "H (7) "E (8) "H (9) "OTTO [1 2 3 4 3 2 1]

CHAPTER 8

BUILDING WORDS AND LISTS

CONCEPTS:

putting words and lists together
randomly picking from a list
haiku poetry

PRIMITIVES:

SENTENCE, LIST, FPUT, LPUT, COUNT, ITEM

8.0 Introduction

How would you like to own a computer that spoke English? Think about how interesting it would be if you could say to the computer, "Print out a graph of costs versus sales for the month of October." Or suppose that you could say, "Find me all the games in 1984 where the Washington Redskins passed more than 40 times," or even, "My car doesn't run very well. Can you help me find out what is wrong."

What would it take to create computers like this? How long might it be before it really happens? Or, you might be thinking, "What are questions like this doing in *this* book anyway?" One answer lies in "lists."

The Logo language has two features not usually found in other languages. These two features are Logo's turtle-graphics and Logo's list handling capability. Just as turtle-graphics provides us with a rich visual play-

ing field, Logo's list handling primitives give us fertile ground for using language.

What makes lists so useful? Two properties. First, lists can contain words or sentences similar in construction to our natural language. And second, we can build up lists of lists, like paragraphs of sentences, or even lists of lists of lists, like chapters of paragraphs of sentences. These two list processing characteristics have helped open the door in Artificial Intelligence (AI) programming to new types of programs that come with such exotic names as "expert systems" and "natural language" programs. A major distinction of these types of AI programs is their ability to "interpret" the English language. Programs that will do what we suggest above are just around the corner. In the next few years, we *will* be able to talk to our computers.

Logo inherited its list handling ability from a language called LISP which derived its name from "*lis*t *p*rocessing." LISP is the most widely used language for Artificial Intelligence programming. Logo's tools for working with lists have come to us from LISP.

We have used the tools which take words and lists apart. Now, let's look at the important topic of constructing new words and lists. Logo provides five primitives, all operations, for doing this; one primitive for putting words together and four primitives for putting lists together: WORD, SENTENCE, LIST, FPUT, and LPUT.

8.1 The Primitive WORD

The primitive WORD, like PRINT, and certain other Logo primitives, comes in two flavors. The usual form of WORD takes two inputs. However, by using parentheses, WORD can take any number of inputs. Here are two simple examples.

```
WORD "FLIM "FLAM
  outputs FLIMFLAM


(WORD "HI "HO "SILVER)
  outputs HIHOSILVER
```

WORD can only take words and numbers as inputs as described in our formal definition.

WORD	**takes two inputs**
WORD	**(takes any number of inputs when used with parentheses)**
Purpose	Joins together two or more inputs into one word
Abbrev	None
Inputs	Two or more inputs, which may be words or numbers
Outputs	The output is a single word

More examples:

```
?PRINT WORD 198 4
1984


?PRINT (WORD "HERE ". "IS ". "ONE ". "WORD)
HERE.IS.ONE.WORD
```

Here is an important use of the primitive WORD to correctly add punctuation to a line.

```
TO PUNCTUATE
PR WORD "WOW "!
END


?PUNCTUATE
WOW!
```

Now that we can put words together, let's tackle lists.

8.2 The Primitives SENTENCE and LIST

The primitives SENTENCE and LIST have a lot in common. Both are used to construct lists from their inputs. Both come in two varieties, like WORD, taking either two inputs or multiple inputs. Both output a list when used. The major difference between them is demonstrated in the way they handle lists as inputs. SENTENCE removes the outer brackets from input lists before combining them while LIST does not. Let's look at SENTENCE first.

```
SENTENCE "HOT "CHOCOLATE
  outputs [HOT CHOCOLATE]


(SENTENCE [COFFEE] [TEA] [OR MILK])
  outputs [COFFEE TEA OR MILK]
```

Notice in the last case, we used parentheses to allow SENTENCE to take three inputs. Notice also, and this is important, that SENTENCE made them into a list of four elements and not three elements. (LIST would have made a list of three elements.)

Here is the formal definition of SENTENCE.

SENTENCE	**(takes two inputs)**
SENTENCE	**(takes multiple inputs when used with parentheses)**
Purpose	Joins together two or more inputs into one list
Abbrev	SE
Inputs	Two or more inputs, which may be numbers, words or lists
Outputs	The output is a single list

Examples:

```
?PRINT SENTENCE "HELLO "BOB
HELLO BOB
```

SENTENCE produces a list even though the outer brackets are not printed when using the PRINT command. For example, if we ask PRINT to print a list

```
?PRINT [THIS IS A LIST]
THIS IS A LIST
```

The brackets are not printed. If you really want to convince yourself that SENTENCE is producing a list, use SENTENCE without a PRINT command before it.

```
?SENTENCE "10 "20
I DON'T KNOW WHAT TO DO WITH [10 20]
```

The two inputs "10 and "20 are made into a list enclosed by brackets. However, we get an error message that Logo doesn't know what to do with the output from SENTENCE. Remember, we always feed outputs to an operation or to a command like PRINT.

```
?PR (SE "NOW "IS "THE "TIME)
NOW IS THE TIME


? PR (SE "AA [BB] [[CC] [DD]])
AA BB [CC] [DD]
```

This last example demonstrates once more how SENTENCE removes the *outer* brackets from any lists it gets as inputs and combines them into one list.

Here is a common usage of SENTENCE. It combines a variable with a word and a list and outputs a new list to PRINT. (Note: Variables inside of lists do not get evaluated. The method below is one way to combine variable values and lists into one list.)

```
TO GREET :NAME
PR (SE "HI WORD :NAME ",[I'M GLAD TO SEE YOU'RE SO HAPPY
 TODAY!])
END

?GREET "JENNY
HI JENNY, I'M GLAD TO SEE YOU'RE SO HAPPY TODAY!
```

Notice how we had to use WORD to string the comma to JENNY. Here is another sample procedure that illustrates how we combine lists with variables:

Sample Procedure: Multiplying Two Numbers

```
TO MULTIPLY :X :Y
PR (SE [THE PRODUCT OF] :X [MULTIPLIED BY] :Y [IS] :X*:Y".)
END

?MULTIPLY 4 5
THE PRODUCT OF 4 MULTIPLIED BY 5 IS 20 .
```

Notice where the period at the end of the sentence wound up. To put the

period directly after the result, 20, you would have to use WORD as in the GREET example. It would begin to look complicated as shown below.

```
TO MULTIPLY :X :Y
PR (SE [THE PRODUCT OF] :X [MULTIPLIED BY] :Y [IS] WORD
 :X*:Y ".)
END

?MULTIPLY 6 7
THE PRODUCT OF 6 MULTIPLIED BY 7 IS 42.
```

Our next example shows how to add space to a printed statement.

```
?PR (SE "EXTRA " " " " "SPACES)
EXTRA    SPACES
```

Now let's see how the primitive LIST works and then compare it with SENTENCE.

If we give the command

```
?PR LIST 10 20
10 20
```

we see that LIST combines its inputs into a list similar to SENTENCE. Here is the formal definition of LIST.

LIST	**takes two inputs**
LIST	**(takes any number of inputs when using parentheses)**
Abbrev	None
Purpose	Joins together two or more inputs into one list
Inputs	Two or more inputs, which may be numbers, words or lists
Outputs	The output is a single list

Examples:

```
?LIST "A "B
I DON'T KNOW WHAT TO DO WITH [A B]

?PR (LIST 10 20 30 40)
10 20 30 40
```

```
?PR (LIST "A 10 [PONDER])
A 10 [PONDER]
```

Here is an example emphasizing the difference between SENTENCE and LIST.

```
?PR (SE [BOSTON] [ST. PAUL] [NEW YORK])
BOSTON ST. PAUL NEW YORK
?
?PR (LIST [BOSTON] [ST. PAUL] [NEW YORK]
[BOSTON] [ST. PAUL] [NEW YORK]
```

We see that SENTENCE does not work here. It takes three lists and combines them into a single list with five elements. LIST, on the other hand, maintains the correct relationship between the inputs by creating a list with three elements.

Let's summarize this section on SENTENCE and LIST with a simple table comparing all the possible inputs and their resultant output.

Table 8-1. Comparing SENTENCE and LIST

	input 1	input 2	output
SENTENCE	"MODULAR	"PROGRAM	[MODULAR PROGRAM]
SENTENCE	"HAPPY	[GO LUCKY]	[HAPPY GO LUCKY]
SENTENCE	[1 2]	[BUTTON SHOE]	[1 2 BUTTON SHOE]
SENTENCE	[3 4]	[CLOSE [THE DOOR]	[3 4 CLOSE [THE DOOR]]
LIST	"MODULAR	"PROGRAM	[MODULAR PROGRAM]
LIST	"HAPPY	[GO LUCKY]	[HAPPY [GO LUCKY]]
LIST	[1 2]	[BUTTON SHOE]	[[1 2] [BUTTON SHOE]]
LIST	[3 4]	[CLOSE [THE DOOR]	[[3 4] [CLOSE [THE DOOR]]]

Whereas SENTENCE and LIST each take two or more inputs and makes lists out of them, our next two primitives each take one data object and add it to an existing list.

8.3 The Primitives FPUT and LPUT

FPUT, which stands up "firstput," and LPUT, which stands for "lastput," are both used to add a Logo data object to an existing list. The only difference between FPUT and LPUT is where the data object is added

to the list. This can be seen in the following two examples and in Fig. 8-1 below.

```
FPUT "A [B C D]
  outputs [A B C D]

LPUT "A [B C D]
  outputs [B C D A]
```

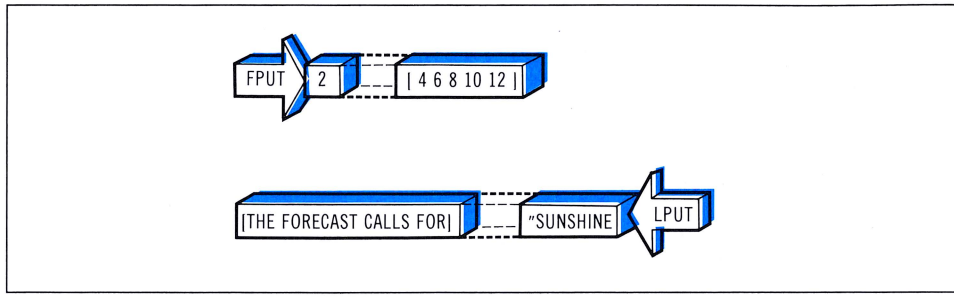


Fig. 8-1. Comparing FPUT and LPUT.

Here is a formal definition of FPUT. Since FPUT and LPUT are so similar, we will not give the formal definition of LPUT.

FPUT	**(takes two inputs)**
Purpose	Adds the first input to the beginning of the list, which must be the second input
Abbrev	None
Inputs	The first input can be any Logo data object The second input must be a list
Outputs	A new list

Examples:

```
FPUT [RED] [YELLOW BLUE [LIGHT GREEN]]
  outputs [RED YELLOW BLUE [LIGHT GREEN]]
```

Since FPUT outputs a list, that output must be passed on to another procedure like PRINT.

```
?PR FPUT "1492. [WHEN THE AMERICAN INDIANS DISCOVERED
 COLUMBUS]
1492. WHEN THE AMERICAN INDIANS DISCOVERED COLUMBUS
```

The primitive LPUT works similarly. Notice that the first input to LPUT is put at the end of the list.

```
?PR LPUT [THE FIRST] [SHALL BE LAST]
SHALL BE LAST THE FIRST
```

Now let's look at how we can use recursion and LPUT to construct a list of squares that look like this

```
[1 4 9 16 25 36]
```

We will first look at a procedure that doesn't quite work the way we want. See if you can figure out the problem.

```
TO ADDLIST :X :LIST
IF :X > 6 [STOP]
PR LPUT :X*:X :LIST
ADDLIST :X+1 :LIST
END

?ADDLIST 1 [ ]
1
4
9
16
25
36
```

This almost does the job. We get our squares all right, but the problem seems to be that we aren't adding to the list; we're just creating new lists, each with one number. Somehow we have to create a new list made up of the old list and the element to be added.

In Logo, the MAKE primitive, formally introduced in the next chapter, is just what we need. MAKE can be used to create variables. For example,

```
MAKE "X 40
```

Here is how we will use MAKE,

```
MAKE "LIST LPUT :X :LIST
```

There are two ways of explaining this line.

1. Make a new list out of the old list and :X.

or

2. Make the variable "LIST equal to the output from LPUT.

Here's another view of this reasoning.

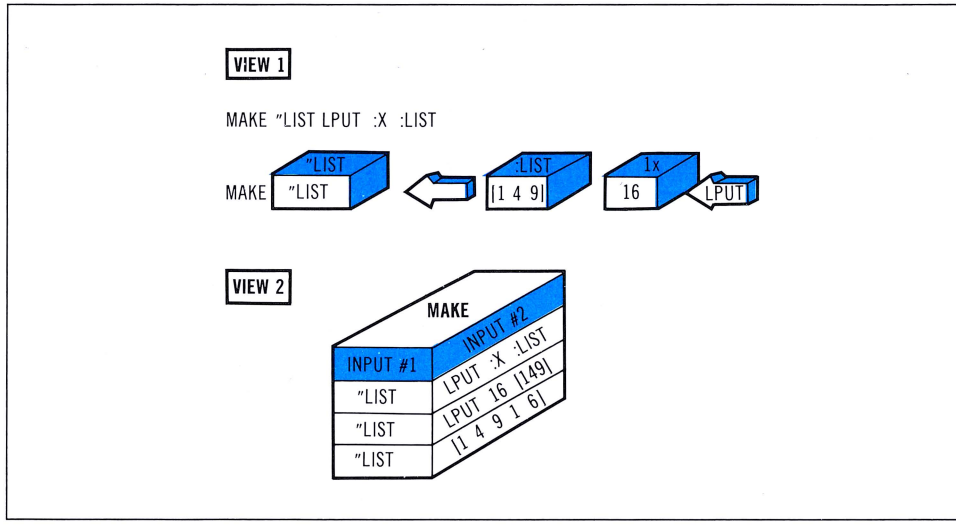


Fig. 8-2. Using LPUT to add to a list.

In the procedure below, we have added a line to use the MAKE primitive to construct bigger lists from smaller lists.

```
TO ADDLIST2 :X :LIST
IF :X > 6 [STOP]
MAKE "LIST LPUT :X *:X :LIST
PR :LIST
ADDLIST2 :X+1 :LIST
END

?ADDLIST2 1 [ ]
1
```

```
1 4
1 4 9
1 4 9 16
1 4 9 16 25
1 4 9 16 25 36
```

Perfect! Recursion is doing the work, creating a list of squares. An easy modification to the procedure is to change the first input to LPUT to :X*:X*:X which would generate a table of cubes. Of course, any other mathematical functions can also be used in place of x-squared.

8.4 Other Word and List Handling Primitives

In addition to the basic primitives that take apart and put together words and lists, Logo provides other primitives which are useful for manipulating lists. These primitives include COUNT and ITEM. These two primitives both work with lists: COUNT counts the number of elements in a list and ITEM outputs one specific element from the list.

After giving a formal definition of each of these primitives, we will create a procedure we call, RANPICK, that selects and outputs a random element from a list. We then conclude this section with an interesting program called HAIKU that picks out phrases from a list and generates random poetry in an Oriental tradition.

8.4.1 The Primitives COUNT and ITEM

The primitive COUNT counts the number of elements in a list. In the following example, COUNT outputs the number "9", since the list contains nine elements:

```
?COUNT [THIS LIST CONTAINS EIGHT WORDS AND ONE NUMBER 1776]
9
```

Here is the formal definition of COUNT.

COUNT	**(takes one input)**
Purpose	Counts the number of elements in a list
Abbrev	None
Inputs	A list
Outputs	Outputs the number of elements in the list

Examples:

```
?PRINT COUNT [10 20 30 40 50 60 70]
7

?PRINT COUNT [FORD PONTIAC CADILLAC PORSCHE HONDA]
5

?PRINT COUNT [[HERE IS ONE LIST] [HERE IS A SECOND LIST]]
2
```

In this last example, COUNT looks at a list with only two elements, each of which is a list with four elements. Since COUNT only counts the outer list, it outputs the number "2".

We will be using COUNT in the procedure shortly, but first we need to see how ITEM works.

The primitive ITEM is used to output a specific element from a list. The element is identified by its location in the list as shown below:

```
?ITEM 3 [THIS LIST CONTAINS FIVE WORDS]
  outputs CONTAINS
```

Here ITEM outputs the third element in the list.

Now for the formal definition of ITEM.

ITEM	**(takes two inputs)**
Purpose	Outputs one element from a list
Abbrev	None
Inputs	A number, say n, and a list
Outputs	Outputs the nth element from the list. If n is greater than the number of elements in the list, Logo signals an error message

Examples:

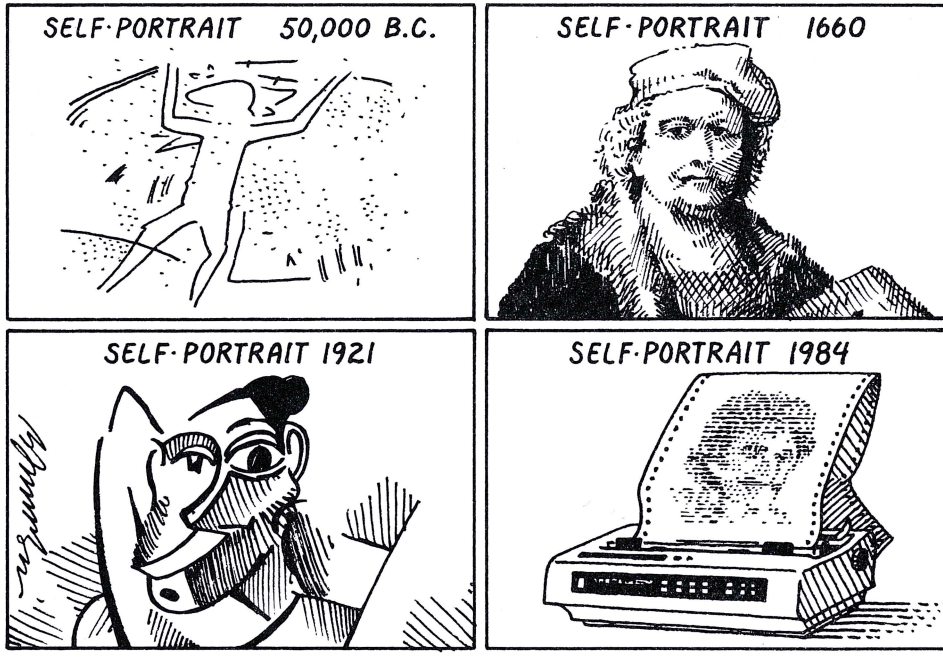
```
?PRINT ITEM 4 [10 20 30 40 50]
40

?PRINT ITEM 2 [[ONE LIST] [TWO LIST] [THREE LIST] [FOUR]]
TWO LIST
```

In the last example, we see that ITEM, like COUNT, works with the outer list only. What happens if we give ITEM incorrect information?

```
?PRINT ITEM 5[A B C]
TOO FEW ITEMS IN [A B C]
```

Logo complains with an appropriate error message that it can't do the impossible; it can't remove the fifth element in a list of three elements.

Now let's put COUNT and ITEM together in a complex, but useful, procedure called RANPICK.



8.4.2 Sample Procedure: Randomly Picking an Element from a List

The procedure RANPICK is short, but powerful. It uses two new primitives that we formally introduce in later chapters, RANDOM and OUTPUT. RANDOM takes one input, say a number "n" and outputs a random number between zero "0", and "n-1". Here are two examples:

```
?PR RANDOM 12
4
```

```
?PR RANDOM 12
9
```

The primitive OUTPUT, as we will see in the next chapter, outputs its input. That is, it is used to create procedures that are operations. We already saw a simple example using OUTPUT in the procedure PI (OUTPUT 3.14159). Now that we know how the four primitives work, let's put them together as shown below:

```
TO RANPICK :LIST
OUTPUT ITEM (1+RANDOM COUNT :LIST) :LIST
END

?RANPICK [1 2 3 A B]
B

?RANPICK [1 2 3 A B]
2
```

How do we analyze this one-line procedure? The secret to analyzing any program is to first identify primitives and their inputs. In this case, all unmarked words are primitives and not procedures. To begin our analysis, we can read the program either from left to right or from right to left. It is slightly easier to explain reading from right to left. The rightmost element in the program :LIST is an input, but what we are looking for are primitives. So moving inside the parentheses, the rightmost primitive is COUNT which we "know" takes one input, in this case the input :LIST (still inside the parentheses). The rest of the analysis is shown below.

Let's assume that :LIST contains [1 2 3 A B], so that the one line in the program

```
OUTPUT ITEM (1 + RANDOM COUNT :LIST) :LIST
```

now looks like this

```
OUTPUT ITEM (1 + RANDOM COUNT [1 2 3 A B] [1 2 3 A B]
                         COUNT [1 2 3 A B]) outputs 5
```

Now mentally substitute the number 5 in place of COUNT [1 2 3 A B] to get

```
OUTPUT ITEM (1 + RANDOM 5) [1 2 3 A B]
RANDOM 5 outputs a number between 0 and 4, say 3
```

Do the same mental substitution again

```
OUTPUT ITEM (1 + 3) [1 2 3 A B]
1 + 3 outputs 4
```

And again

```
OUTPUT ITEM [1 2 3 A B]
ITEM 4 outputs "A
```

Now the "A" is fed to OUTPUT

```
OUTPUT "A
  outputs "A
```

Thus, the procedure RANPICK [1 2 3 A B] outputs "A to become an input to another primitive or procedure.

Notice how each primitive in this example is an operation that outputs a data object to the next operation and so on.

Here is a helpful little procedure for those who have trouble making up their minds.

```
TO ICE.CREAM
MAKE "LIST [CHOCOLATE VANILLA STRAWBERRY [ALMOND MOCHA]
[ROCKY ROAD]]
PRINT RANPICK :LIST
END

?ICE.CREAM
VANILLA

?ICE.CREAM
ALMOND MOCHA
```

In this procedure, we use the MAKE primitive to create a list that we call :LIST and then feed the list to RANPICK. There are several possible extensions of this procedure that you might try. For example, we could use SENTENCE to write more about our choices. Later, we'll see how to construct the list of ice creams interactively from the keyboard.

Now let's take a more interesting example where we construct random sentences, or in this case, Japanese-style poems.

8.5 Sample Program: Haiku Poetry

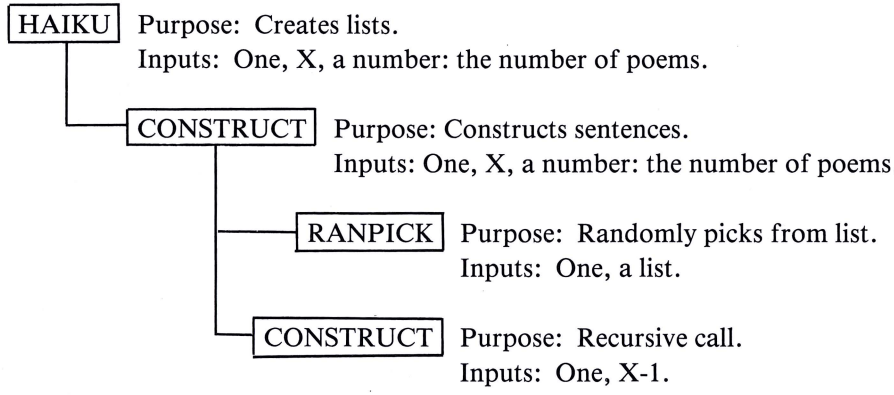
Haiku is a form of Japanese poetry that produces richly suggestive imagery using simple, yet highly structured form. Since the Japanese language is so different from English, translations and new creations of Haiku are very pale imitations of the original art. Here is a classic example of Haiku from the book, "An Introduction To Haiku," by H. G. Henderson.

Old pond:
frog jump-in
water-sound.

Our Haiku poetry generator has five lists that can be put together in different ways. The basic form is like this.

```
RANPICK A, RANPICK FIRST B
 RANPICK C, RANPICK D
  RANPICK E, RANPICK LAST B
```

A, C, D, E, are lists, each list containing three words or phrases for RANPICK to choose from. The list B has two lists, each storing three elements that allows us to rhyme the end of the first and third lines. Here is the box-chart for the program showing each procedure.



Here is a sample run of the program.

```
?HAIKU 2
SOFT SPRING DEW
```

```
HAWKS SOARING, FRESH BREEZES
 EVENING BLUE

WARM SUMMER DREAM
 SURF SUNNING, MISTY CLOUDS
  TWILIGHT GREEN
```

The listing of the procedures is as follows:

```
TO HAIKU :X
MAKE "A [COOL SOFT WARM]
MAKE "B [[[SPRING DEW] [BLUE]] [[SUMMER DREAM] [GREEN]]
[[SNOW-CAPPED HEADS] [RED]]]
MAKE "C [[FLOWERS WAKING,] [SURF SUNNING,] [HAWKS
 SOARING,]]
MAKE "D [[MISTY CLOUDS] [FRESH BREEZES] [SWOLLEN RIVERS]]
MAKE "E [DAWN TWILIGHT EVENING]
CONSTRUCT :X
END

TO CONSTRUCT :X
IF :X =0 [STOP]
MAKE "B1 RANPICK :B
PR (SE RANPICK :A FIRST :B1)
PR (SE " " " RANPICK :C RANPICK :D)
PR (SE " " " " " " RANPICK :E LAST :B1)
CONSTRUCT :X-1
END
```

Since there are five lists created with the MAKE primitive, and each containing three elements to pick from, there are 125 possible different poems (5 to the 3rd power= 125). The input, :X, lets you choose the number of poems you want written.

It is easy to improve the poems by substituting or adding new words to one or more lists.

Now let's summarize what we have learned.

8.6 Summary

Here are the important points in this chapter.

1. Numbers, words and lists can be put together with the operations SENTENCE, LIST, FPUT, and LPUT.

2. SENTENCE combines its inputs to form a list. The outer brackets of its list inputs are not kept.
3. LIST combines its inputs to form lists, but LIST keeps the outer brackets of list inputs.
4. Lists can be also manipulated with the operations COUNT and ITEM.
5. RANPICK is a useful utility that randomly picks an element from a list.

8.7 Review Questions and Exercises

Review Questions

1. Give an example that shows how SENTENCE and LIST produce different results.
2. How can you print the following statement, where X must be evaluated by Logo, i.e., we use :X
   THE VALUE OF X IS 5!
3. Can FPUT put lists into lists as well as words into lists?
4. Can COUNT count letters in a word?
5. Will ITEM output a character from a word as well as an element from a list?
6. What will the following produce?
   PRINT SUM COUNT :LIST ITEM 4 :LIST
   when the LIST contains [A 3 R 5 OK4U]

Exercises at the Terminal

1. Create a procedure that simulates the rolling of dice and prints out the result.
2. Create a procedure that constructs lists of "cubes."
3. Create a procedure that prints random sentences like
   "THE SCRAGGLY DOG AMBLED SLOWLY DOWN THE STREET."
   Use randomly chosen noun phrases, verbs, adverbs and prepositional phrases.

Answers to Review Questions

(1) SENTENCE "ENGLAND [WEST GERMANY] outputs [ENGLAND WEST GERMANY], while LIST "ENGLAND [WEST GERMANY] outputs [[ENGLAND [WEST GERMANY]] (2) PR (SE [THE VALUE OF X IS] WORD :X "!) (3) Yes (4) No (5) No (6) 10

CHAPTER 9

VARIABLES: INPUTS, OUTPUTS AND THE MAKE PRIMITIVE

CONCEPTS:

creating variables three ways
giving values to inputs
using MAKE statements
local versus global variables
writing interactive programs
comparing inputs with the MAKE command
using OUTPUTs as variables
recursion with OUTPUTs

PRIMITIVES:

MAKE, READLIST, LOCAL, OUTPUT

9.0 Introduction

Did you realize that your car is a variable speed vehicle? Or that you consume a variable number of calories each day? Or that music involves sounds of variable frequencies? We live in a variable world, a world of changing values. The world would not be as interesting a place if it were not for these changing values; variability is the substance and the spice of life.

Any computer language worth its salt must be able to handle variables if it is to be useful to us in such a dynamic world. Logo is quite adept at han-

dling variables. It gives us three ways to create them. We can use inputs, MAKE statements, or procedures with OUTPUTs.

Here is a simple example of each:

Using Inputs to Create Variables

```
?TO GO.FAST :SPEED
>PRINT :SPEED
>END

?GO.FAST 70
70
?
```

We give the variable a value when we call the procedure

Using MAKE to Create a Variable

```
?MAKE "SPEED 70
?PRINT :SPEED
70
?
```

Using OUTPUT to Create a Variable

```
?TO SPEED
>OUTPUT 70
>END

>PRINT SPEED
70
?
```

In each case, we had a variable name, "SPEED," and a variable value, 70. (Remember that one of the "Cardinal Rules of Good Programming Etiquette" is to choose meaningful names for variables. It may seem easy and convenient at the time to use "GO.FAST :X," but if the name reduces the readability of the program, don't use it. This is especially important if a variable number of months may pass before you use the program again. To paraphrase an old adage: "a good name is worth its weight in gold.")

Consider our three GO.FAST procedures. Which one is best? Which method should we use in writing procedures? It depends. In this chapter, we'll explore the circumstances that determine which method you choose for creating variables.

We'll begin this chapter by discussing inputs, then we'll compare inputs with MAKE statements. Towards the end of this chapter, we'll define OUTPUT and give several examples of its use. As we go through the different methods of programming with variables, we'll point out the "pros and cons" of using each method. As with other aspects of Logo, you'll find you have considerable freedom of choice, depending on what it is you want to do!

9.1 Inputs

We've used inputs frequently in the last few chapters. You may want to review the discussion of inputs in Chapter 3, especially sections 3.1.2 ("Five Kinds of Inputs").

We've seen, in formal definitions and in programming examples, that most of Logo's primitives take one or more inputs. Although we haven't made the distinction until now, inputs for primitives do not have names associated with them. For example, FORWARD takes a number input, but we don't have a name for it.

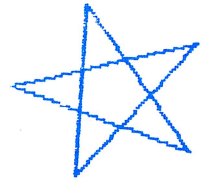
However, when we write procedures, we must give inputs a name. We've seen that such procedures have the input names on the title line.

Once named, the input (as a dotted word) can be used in the body of the procedure. For example:

```
?TO STAR :SIZE
>REPEAT 5 [FD :SIZE RT 144]
>END
```

This input name, as a dotted word, passes its value to FORWARD.

```
?STAR 80
```

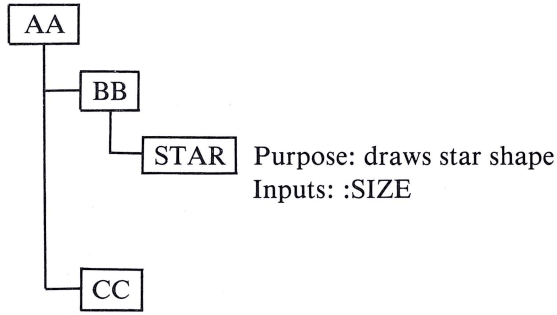


The fundamental question concerning procedures with inputs is "where do the inputs get their value?" Aha, you say, I know where they get their

value. I provide the input value whenever I run the procedure. For example, I can direct Logo to

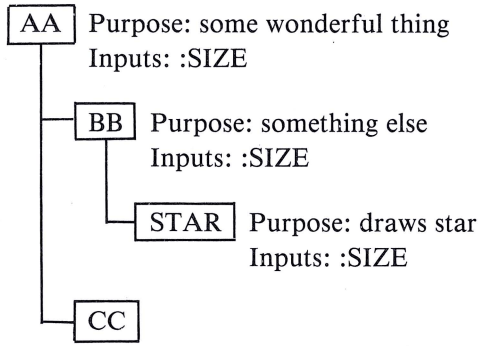
```
STAR 30
STAR 45
STAR 16.672
```

And you're right! In the case of STAR above, that's exactly where the input gets its value. However, let's look at a more general case where a procedure is called by another procedure, as shown in the box chart below. This program is made up of procedures AA, BB, STAR, and CC. Procedure AA calls BB which calls STAR; when STAR ends, AA calls CC. Don't worry about what these mysterious alphabetical procedures do; we're just showing a hypothetical situation:



Now how does STAR :SIZE get its input value? There are four possible solutions to the case of input supply. We'll outline these methods briefly, then discuss them fully later in the chapter.

(1) The first way we can assign values to inputs is through a method we used earlier. We add :SIZE to the title line of procedure AA and BB and just pass the value along when we call procedure AA. For example:



Can we do that even though procedure AA doesn't use :SIZE? The answer is "Yes." In fact, we don't have to include :SIZE in BB; STAR will find it in AA. What about CC—does it need to have :SIZE in its title line? No, by the time the procedure has gotten to CC, STAR has ended and :SIZE isn't needed.

We've used this method of passing values to inputs before and found that it works quite well. The only disadvantage is that we clutter up the title lines with another input variable. Suppose procedure AA, for example, has five inputs of its own; all that procedure line doesn't need is one more input!

(2) A second way STAR can get its input value is to assign the value in the procedure that calls STAR. For example, the program code of BB might look like this:

```
?TO BB
>- - - - - -
>- - - - - -
>STAR 15
>- - - - - -
>END
```

Now when we run the program, AA calls BB which calls STAR and assigns the value of 15 to :SIZE automatically. The advantage of this method is that it's very simple. But, as you may have guessed, if we wanted to make a larger (or smaller) star, we'd have to edit BB. We lose flexibility using this method of passing values to inputs.

(3) A third way of passing a value to an input is with the primitive MAKE. We'll discuss MAKE in more detail in the next section. Briefly, MAKE takes a quoted word input and assigns it the value of the second input, as in the example below:

```
MAKE "SIZE 15
```

To continue using our mythological program AA, the MAKE command could be placed as a line of code in procedure AA or BB. This method is quite similar to method #2 above.

```
?TO BB
>- - - - - -
>- - - - - -
>MAKE "SIZE 15
>STAR :SIZE
>END
```

The advantage of the MAKE command used in this way is that it is simple. The disadvantage is the same as with method #2, i.e. in order to change the star's size, we would need to edit BB.

(4) There is another way to use the MAKE command to get around this last problem. We'll combine the MAKE command with the primitive READLIST. READLIST is formally introduced in section 9.4, but let's take a sneak preview at it now. We substitute READLIST for the input value in MAKE, and ask the user to decide what they want the size of the star to be. The following lines of code are from such a procedure:

```
>PR [WHAT WOULD YOU LIKE THE SIZE OF THE STAR TO BE?]
>MAKE "SIZE FIRST READLIST
>STAR :SIZE
```

When this portion of the program is run, here is what happens:

(a) The question "WHAT WOULD YOU LIKE THE SIZE OF THE STAR TO BE?" would appear on the screen; then READLIST causes the program to pause until a number and RETURN are typed by the user.

(b) the number typed is placed in a list by READLIST. The primitive, FIRST, removes the number from the list and outputs it to MAKE.

(c) MAKE assigns the number to the variable named SIZE.

MAKE, used with READLIST, is a very useful way of writing programs and procedures where you'd like to have a friendly exchange between the computer user and Logo. This exchange can be used to give inputs their values. Let's take a closer look at this important Logo primitive.

9.2 The MAKE Primitive

The primitive MAKE is widely used both to create variables and to change variable values. MAKE can be used inside of procedures as we've seen above or at the Command Level. When used with READLIST, or its relative, READCHAR, MAKE lets us create interactive programs. Finally, MAKE can create a "global" variable, available to all primitives and procedures or it can be used to create a "local" variable, one restricted to the

procedure creating it. Let's formally define MAKE and then explore these characteristics.

9.2.1 Using MAKE Statements

MAKE takes two inputs, the first input is a word, the name of the variable. The second input becomes the value to be associated with the first. For example,

```
?MAKE "X 15
?PRINT :X
15
?
```

Here are some other examples of MAKE statements:

```
?MAKE "FOOD [APPLES, ORANGES, FIGS, CHERRIES]
?PRINT :FOOD
APPLES, ORANGES, FIGS, CHERRIES
?

?MAKE "PERSON "JOE
?PRINT :PERSON
JOE
?

?MAKE "X :X + 10
?PRINT :X
25
?
```

The last example assumes that the old value of X was 15.
Here is the formal definition of MAKE:

MAKE	**(takes two inputs)**
Purpose	Assigns a value to a variable
Abbrev	None
Inputs	Two: first must be a word, second can be a number, word, list, dotted word or operation
Outputs	None

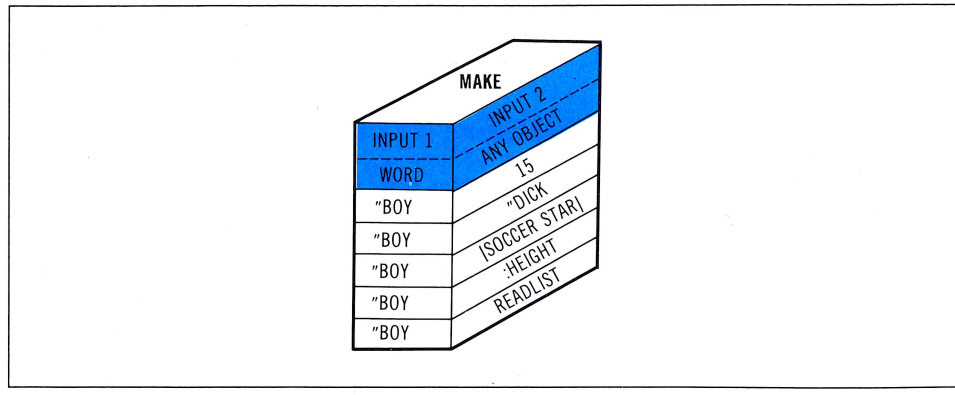


Fig. 9-1. The MAKE primitive.

Let's look at the most common uses of MAKE: creating and changing variables.

Here is a simple example that uses MAKE to create a list and then uses the utility RANPICK (from Chapter 7) to output an element from the list. Our example is in the form of a thesaurus.

```
?TO THESAURUS
>MAKE "JOVIAL [AMIABLE AFFABLE JOCULAR JOLLYING]
>MAKE "CREATE [FORM CONSTRUCT MAKE DESIGN]
>END

?TO REPLACE :WORD
>PRINT RANPICK :WORD
>END

?THESAURUS
?REPLACE :JOVIAL
AFFABLE
?
?REPLACE :CREATE
DESIGN
```

Let's analyze this by considering inputs. The procedure REPLACE is defined with one input called WORD. However, REPLACE really wants a list as input since RANPICK requires a list as input. When we call REPLACE with :JOVIAL as input, the value of JOVIAL is a list created in THESAURUS.

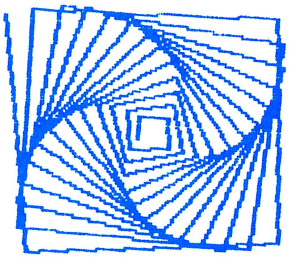
A good programming exercise would be to rewrite REPLACE so that you can choose the replacement word rather than randomly picking it. This would give us an interactive Thesaurus.

Another common use for MAKE is to change the values of a variable. For example, we can write a spiral procedure that uses MAKE and REPEAT in place of recursion. It would look like this:

```
TO SPIRAL.MAKE
MAKE "LEG 10
REPEAT 50 [FD :LEG RT 88 MAKE "LEG :LEG + 2]
END
```

We start with an initial value for LEG and let MAKE increase it with each REPEAT. Here is the result.

```
?SPIRAL.MAKE
```



Generally, we prefer to use recursion with changing inputs rather than use the MAKE primitive as shown above. The recursive program is simpler to write and easier to read. We also choose recursion to give us practice in using this powerful programming tool. Also, variables created using inputs are "local" to the procedure, whereas, MAKE, as used above, creates a "global" variable.

However, MAKE can be used to create local variables too; let's look at this in more detail.

9.2.2 Local and Global Variables

What is a local variable? What is a global variable? Which is preferred? If you have typed in the procedures we've used in this chapter and you now check the workspace titles by typing POTS, here is what you get.

```
?POTS
GO.FAST :SPEED
```

```
SPEED
STAR :SIZE
THESAURUS
REPLACE :WORD
SPIRAL.MAKE
```

Since we've run these procedures, let's see if :SIZE has a value by typing

```
?PR :SIZE
SIZE HAS NO VALUE
```

Even if we were to run the STAR procedure again and give :SIZE a value of 100 by typing

```
?STAR 100
```

we still get an error message if we type PR :SIZE at the Command Level.

The reason for the error message is that :SIZE is a local variable. Its value (temporarily 100 in the above case) is known only to the procedure STAR and any subprocedure it calls. When the procedure has ended, :SIZE loses its value.

Let's see if we have any global variables. One way to find out is to use PRINT at the Command Level like before. We could type

```
?PR :SIZE
SIZE HAS NO VALUE
```

Thus, SIZE is not a global variable.

```
?PR :LEG
LEG IS 50
```

Aha, :LEG is a global variable. Any variable whose value is known at the Command Level, we call global. A global variable makes its value available to all primitives and procedures.

Is there an easier way to find global variables rather than trying to remember their names and using PRINT? Yes. Since global variables reside in the workspace, they can be seen with the primitive POALL (Print out all) or PONS (Print out names). POALL prints out all procedures and global variables. PONS just prints out global variables. Suppose we type PONS and get the following:

```
?PONS
LEG IS 110
```

```
SPEED IS 70
FOOD IS [APPLES, ORANGES, FIGS, CHERRIES]
PERSON IS JOE
JOVIAL IS [AMIABLE AFFABLE JOCULAR JOLLYING]
CREATE IS [FORM CONSTRUCT MAKE DESIGN]
```

Look at that! All those variables created by MAKE earlier are still there. Even if we don't need them anymore, they are stored in the workspace. To use a global variable stored in the workspace, just type its name preceded by the colon.

```
?PR :PERSON
JOE

?PR SUM :SPEED :LEG
185
```

Or we could call the procedure STAR like this, STAR :SPEED, and Logo will use the number 70 (:SPEED) to draw a STAR.

How do we create local and global variables? Here are two rules of thumb.

1. *Inputs* to procedures always create local variables.
2. The *MAKE* primitive creates global variables with two exceptions. The exceptions are

   a. When an input to a procedure creates a local variable and MAKE within the procedure changes the value of the input, the input is still a local variable. For example,

```
COUNTTWOS :X
PR :X
MAKE "X :X + 2     X is still local
COUNTTWOS :X
END
```

   b. When we precede MAKE with the primitive LOCAL, we can create a local variable. For example,

```
TO GETINPUT
LOCAL "ANSWER                          ANSWER is local
```

```
MAKE "ANSWER READLIST
IF :ANSWER = [YES] [PRINT [THANK YOU]]
END
```

All other uses of the MAKE primitive create global variables that can be seen with PONS. In order to get rid of global variables from the workspace, we can use the primitive ERNS which stands for "Erase All Names." "Name" in Logo means the same as variable. If we want to get rid of one global variable, we use the command ERN "variablename, where you place the name of the variable here.

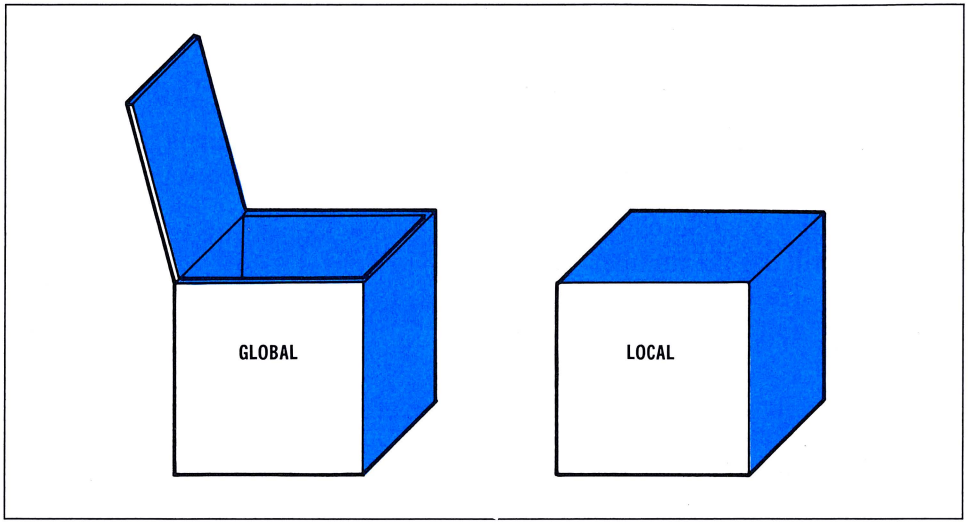


Fig. 9-2. Global variables are available to all procedures.

Now that we have a choice of establishing variables globally or locally, which way is best?

9.2.3 Three Reasons to Go Local

Our title line may give us away . . . most of the time, it's preferable to establish variables locally, as opposed to globally. There are three reasons for this, as follows:

1. Local variables have fewer chances of "bugs" developing when you're assigning variables of the same name in different procedures.
2. Procedures are more modular or "stand alone" if they are written

with variables assigned locally. The contact with the outside world is through inputs (which are local) and outputs only. Procedures within large programs can be deleted, moved or added to without worrying about variable names.

3. The workspace is not cluttered up storing variable values if you go local.

There are, however, times when establishing global variables is a good idea. We'd probably want to "go global" when

1. We want two or more procedures to have access to the same variable, and we don't want to use inputs.
2. We want to save the values of a variable when we save the workspace.

Now that we see how MAKE works, let's take another look at READLIST.

9.3 Interactive Programs: Using MAKE with READLIST

We've seen an example of the primitive READLIST earlier. It is a useful way of assigning values to inputs. READLIST allows for a dialogue or exchange between the computer user and Logo. It allows for flexibility in programs, and makes the running of programs "user friendly"!

For example, let's write a procedure that creates a beginning "conversation" between a computer user and Logo. The first thing we might do is to GREET the user:

```
?TO GREET
>PRINT [WHAT IS YOUR NAME?]
>MAKE "ANSWER READLIST
>PR SENTENCE [NICE TO MEET YOU,] :ANSWER
>PR SENTENCE [I'VE ENJOYED TALKING TO YOU,] :ANSWER
>END
```

When we run GREET, the following is displayed:

```
WHAT IS YOUR NAME?
LUKE SKYWALKER
```

```
NICE TO MEET YOU, LUKE SKYWALKER
I'VE ENJOYED TALKING TO YOU, LUKE SKYWALKER
?
```

READLIST, together with MAKE, then, allows for interactive programs. Here is the formal definition of READLIST:

READLIST	**(takes one input)**
Purpose	Temporarily stops the procedure until keyboard input is given with a RETURN
Abbrev	RL
Inputs	One, whatever is typed from keyboard
Outputs	Outputs whatever the user types in as a list

Let's expand the GREET program to give another example of READLIST and also to continue our discussion of local variables discussed in the last section. We'll modify the GREET program to include a question about the user's home state:

```
?TO GREET
>PR [WHAT IS YOUR NAME?]
>MAKE "ANSWER READLIST
>PR SENTENCE [NICE TO MEET YOU, ] :ANSWER
>STATE
>PR SENTENCE [I'VE ENJOYED TALKING TO YOU, ] :ANSWER
>END
```

The procedure called STATE looks like this:

```
?TO STATE
>PR SENTENCE [I'D LIKE TO KNOW MORE ABOUT YOU, ] :ANSWER
>PR [WHAT STATE DO YOU LIVE IN?]
>MAKE "ANSWER READLIST
>PR SENTENCE :ANSWER [IS A GREAT STATE!]
>END
```

Here is a sample run of the GREET program.

```
?GREET
WHAT IS YOUR NAME?
```

```
LUKE SKYWALKER
NICE TO MEET YOU, LUKE SKYWALKER
I'D LIKE TO KNOW MORE ABOUT YOU, LUKE SKYWALKER
WHAT STATE DO YOU LIVE IN?
ARIZONA
ARIZONA IS A GREAT STATE!
I'VE ENJOYED TALKING TO YOU, ARIZONA
```

Oops! The last line contains a bug. Instead of returning to our first value for :ANSWER, the second READLIST has assigned a new value for :ANSWER—ARIZONA.

There are two ways to solve this problem. One would be to assign the variable associated with STATE a different name from the one we used for the individual's name. For example, we could edit STATE to use a variable name ANSWER2 as follows:

```
?TO STATE
>PR SENTENCE [I'D LIKE TO KNOW MORE ABOUT YOU, ] :ANSWER
>PR [WHAT STATE DO YOU LIVE IN?]
>MAKE "ANSWER2 READLIST
>PR SENTENCE :ANSWER2 [IS A GREAT STATE!]
>END
```

Now when we run the program, things work out much better:

```
?GREET
WHAT IS YOUR NAME?
LUKE SKYWALKER
NICE TO MEET YOU, LUKE SKYWALKER
I'D LIKE TO KNOW MORE ABOUT YOU, LUKE SKYWALKER
WHAT STATE DO YOU LIVE IN?
ARIZONA
ARIZONA IS A GREAT STATE!
I'VE ENJOYED TALKING TO YOU, LUKE SKYWALKER!
```

A second way of correcting the problem would be to use the primitive LOCAL. LOCAL makes the variables local to only that procedure, or any it calls. We could edit STATE to look like this:

```
?TO STATE
>LOCAL "ANSWER
>PR SENTENCE [I'D LIKE TO KNOW MORE ABOUT YOU, ] :ANSWER
```

```
>PR [WHAT STATE DO YOU LIVE IN?]
>MAKE "ANSWER READLIST
>PR SENTENCE :ANSWER [IS A GREAT STATE!]
>END
```

This program works as you will see if you run it. The primitive LOCAL keeps the variable ANSWER in STATE from being used outside of the STATE procedure. Whether you assign different variable names, or use the primitive LOCAL is largely a matter of programming needs.

However, let's say we wanted to change GREET so that the last sentence would be displayed as follows:

```
I'VE ENJOYED TALKING TO YOU, LUKE SKYWALKER; YOU'RE THE
FIRST PERSON I EVER TALKED TO FROM ARIZONA
```

Here, we'd want to use the two variable approach of "ANSWER and "ANSWER2 for LUKE SKYWALKER and ARIZONA, respectively. We have to do this because we are using STATE's answer and GREET's answer both in the same procedure. But in cases where the variable will only be used in a separate procedure, and not referred to again, LOCAL is the best choice.

Here our formal definition of LOCAL:

LOCAL **(LOCAL)**	**takes one input** **(takes multiple inputs if contained within parentheses)**
Purpose	Makes its inputs local to the procedure and any subprocedure
Abbrev	None
Inputs	Variable name
Outputs	None

Now that we have seen two ways to create variables, let's compare them further.

9.4 Comparing Inputs with MAKE

In this section, we'll return to an old familiar program, the SPIRAL program, and compare different methods of assigning values to variables. We'll show four different ways to do this.

9.4.1 SPIRAL1: Using Inputs to Assign Variable Values

Our first version of spiral uses inputs to assign values to variables. The procedure listing for SPIRAL1 looks like this:

```
?TO SPIRAL1 :SIZE :ANGLE :INCREMENT
>FORWARD :SIZE
>RIGHT :ANGLE
>SPIRAL1 (:SIZE + :INCREMENT) :ANGLE :INCREMENT
>END
```

To run this procedure, we need to assign values to :SIZE, :ANGLE and :INCREMENT. We can run SPIRAL1 with values such as those below:

```
?SPIRAL1 1 270 4
?SPIRAL1 50 240 6
```

The advantage of SPIRAL1 is that we can change inputs easily.

Instead of using inputs, let's write a SPIRAL procedure using the MAKE primitive.

9.4.2 SPIRAL2: Using MAKE to Assign Variable Values

Here is a spiral program that does not use inputs. Instead it uses the MAKE primitive to create variables with changing values.

```
?TO SPIRAL2
>MAKE "SIZE 10
>MAKE "ANGLE 44
>MAKE "INCREMENT 2
>CS
>SPIRAL2A
>END

?TO SPIRAL2A
>FORWARD :SIZE
>RIGHT :ANGLE
>MAKE "SIZE (:SIZE + :INCREMENT)
>SPIRAL2A
>END
```

The advantage of SPIRAL2 is that we can run it without remembering and entering inputs. The disadvantage is that it is not as flexible as SPIRAL1.

You might be wondering if we can combine inputs with the MAKE primitive. Yes and here is the result.

9.4.3 SPIRAL3: Combining Inputs with the MAKE Primitive

In the procedure below, we've included both inputs and MAKE commands into a single graphics program:

```
?TO SPIRAL3
>MAKE "SIZE 10
>MAKE "ANGLE 99
>MAKE "INCREMENT 3
>CS
>SPIRAL1 :SIZE :ANGLE :INCREMENT
>END
```

The advantage to this approach is that we can still use SPIRAL1 by itself or as a part of SPIRAL3. Do we have the best of both worlds? Not quite.

There is another way of writing the SPIRAL program. Although it is the most complex method of the four we've used, it has one additional advantage over the others. It is an "interactive" SPIRAL program.

9.4.4 SPIRAL: Creating an Interactive Graphics Program

Suppose you want to have someone who has never used a computer, or who is not familiar with Logo, to see your SPIRAL procedure. Telling them to "Type in SPIRAL1 2 16 30" probably wouldn't make much sense to them. Similarly, someone unfamiliar with Logo wouldn't be able to respond to "Change the MAKE values for :SIZE, :ANGLE, and :INCREMENT by editing SPIRAL2 or SPIRAL3."

Ideally, a procedure that could be used by a novice would give simple directions for allowing values to be assigned. It's up to you, the Logo programmer, to allow those values to be passed on to the right primitives.

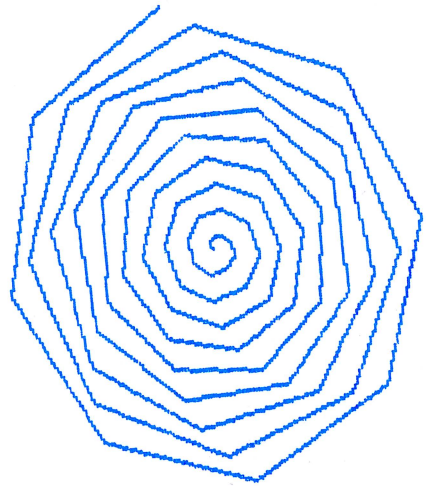
Let's write an interactive SPIRAL procedure that invites the user to enter the values to be used as inputs:

```
?TO SPIRAL
>(LOCAL "SIZE "ANGLE "INCREMENT)
>PR [ENTER THE VALUE FOR THE SIZE?]
>MAKE "SIZE FIRST READLIST
>PR [ENTER THE VALUE FOR THE ANGLE?]
>MAKE "ANGLE FIRST READLIST
>PR [ENTER THE VALUE FOR CHANGING THE SIZE?]
```

```
>MAKE "INCREMENT FIRST READLIST
>CS
>SPIRAL1 :SIZE :ANGLE :INCREMENT
>END
```

Here is a sample run:

```
?SPIRAL
ENTER THE VALUE FOR THE SIZE?
0
ENTER THE VALUE FOR THE ANGLE?
44
ENTER THE VALUE FOR CHANGING THE SIZE?
1
```



Look at how easy it is to use this program! The user is prompted for each specific input. Then, the program uses our original procedure, SPIRAL1, to create a beautiful design.

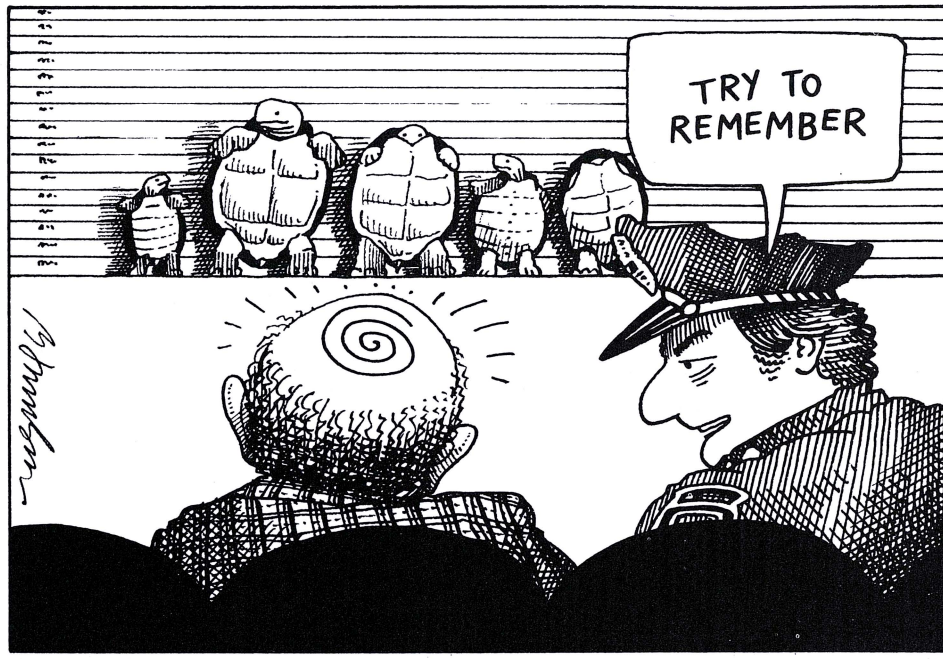
There are ways to make this program even more "user friendly." We could give the user a typical range of values to choose from, as shown in the partial listing below.

```
PR [ENTER THIS VALUE FOR THE SIZE]
PR [TYPICAL VALUES: 1 TO 10]
```

Now our program gives friendly advice. This is the way programs should be written. Another refinement we can add is to provide "error trapping" to insure that the user only enters appropriate numbers. We have written

some "interactive utilities," such as GETB.NUMBER, that provide error trapping when using READLIST. These utilities are introduced in Chapter 12 and Chapter 13.

You might be wondering why we need to insert FIRST before READLIST in the third, fifth, and seventh lines of code? Remember that READLIST outputs the information typed in as a list. Primitives like FORWARD, RIGHT and plus (+) will not accept lists as inputs: they require inputs to be numbers.



SPIRAL allows for the greatest interaction between a user and the computer. You might want to add to it by first greeting the user, and then explaining what the program will do. You may also want to redo some earlier graphics programs that we've written to allow for similar exchange and save these new programs to disk.

Now let's move on and look at one more way to create variables.

9.5 Creating Variables with OUTPUT

In Chapter 3 we saw how Logo primitives can be either commands or operations. Operations output a value. For example:

```
?SUM 3 4
   outputs a 7


?FIRST "BINGO
   outputs B
```

Operations are always used as inputs. They must be fed to a command like PRINT or to another operation.

The primitive, OUTPUT, is used to create *procedures* that are operations. We've seen one of the simplest examples.

```
?TO PI
>OUTPUT 3.14159
>END
```

We also started this chapter with an example called, SPEED, that outputs a number, 70, in that case.

Since operations always become inputs, operations create variables. However, since operations created with OUTPUT are procedures, the variables created can be the result of complex programs. Operations produce a very powerful type of input or variable. You might think of some operations as being "intelligent variables," at least compared to those created by inputs and MAKE.

Let's formally define OUTPUT and then look at some examples.

OUTPUT	**(takes one input)**
Purpose	Stops the current running procedure and outputs its input
Abbrev	OP
Inputs	Any data object
Outputs	Any data object

Here are more examples using OUTPUT.

```
?TO GETA.NUMBER
>OUTPUT FIRST READLIST
>END
```

Notice how this works. The primitive READLIST causes Logo to pause until the user enters something from the keyboard concluded with a

RETURN. Suppose the number 70 is entered. Then, FIRST takes 70 out of the list and feeds it to OUTPUT. OUTPUT feeds 70 to whatever procedure called GETA.NUMBER and then STOPS GETA.NUMBER.

How do we use GETA.NUMBER? Here is a partial example

```
PR [ENTER THE VALUE FOR SIZE]
MAKE "SIZE GETA.NUMBER
```

GETA.NUMBER outputs a number to be used by MAKE. GETA.NUMBER might be used three times in our SPIRAL program. It is so useful, we will call it a utility and add it to our utility disk. We call this utility GETA.NUMBER because later we will create a more sophisticated version called GETB.NUMBER in Chapter 12.

Here is another utility using OUTPUT; we've used it before, too.

```
?TO RANPICK :LIST
>OUTPUT ITEM (1 + RANDOM COUNT :LIST) :LIST
>END

?PRINT RANPICK [1 2 3]
2
```

Notice how RANPICK strings together primitives to do a job.

We can do the same thing with procedures. It is possible to use OUTPUT to create procedures that are "building blocks," each procedure doing part of a complex task.

Here are two examples, one short and one long. First, the short one, an operation that averages two numbers.

```
?TO AVE :N1 :N2
>OP (:N1 + :N2)/2
>END

?PRINT (AVE 10 15) + (AVE 20 25)
35
```

Notice how the output of AVE is fed to plus (+) whose output is fed to PRINT. We have strung three operations together.

For our next example, imagine you're the president of a small company, and you want to calculate the paychecks of your four employees for a given week.

Gross paycheck amounts are calculated by multiplying the number of hours worked times the hourly rate of pay. A simple procedure for obtaining an employee's gross salary would be as follows:

```
?TO PAYCHECK1 :HOURS :RATE
>MAKE "AMOUNT :HOURS* :RATE
>OUTPUT :AMOUNT
>END
```

The amount calculated must be passed to a Logo primitive like PRINT. When we run PAYCHECK1, then, we'll include PRINT on our command line:

```
?PRINT PAYCHECK1 45 8
360
```

What is the reason for using OUTPUT? The advantage of the OUTPUT statement in PAYCHECK1 is that we can use that procedure to perform other calculations. For example, we can direct Logo to calculate the total payroll for all four employees, as follows:

```
?PRINT (PAYCHECK1 40 5)+(PAYCHECK2 35 7)+(PAYCHECK3 42
8)+(PAYCHECK4 30 6)
961
```

and receive the total payroll amount, $961.00.

These are fairly straightforward examples. Let's step up one level in complexity and look at what is often called Logo's most useful and most powerful programming tool, recursion with outputs.

9.5.1 Using OUTPUT in Recursive Procedures

Here is a problem. Suppose you want to add together the numbers, 1, 2 and 3. We know that the total is 6, but can Logo do the job? Here is one way.

```
PRINT 1 + 2 + 3
```

or

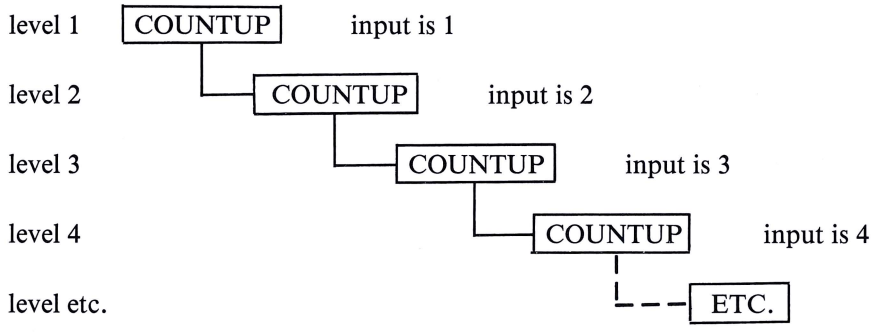
```
PRINT (SUM 1 2 3)
```

However, suppose we want to add up the numbers from 1 to 40. Or 1 to

100. Is there a better, easier way to do that? Of course! We'll use recursion. Do you remember one of the first recursive procedures we tried?

```
?TO COUNTUP :X
>PR :X
>COUNTUP :X + 1
>END
```

Each COUNTUP procedure calls another COUNTUP procedure at a lower level as shown in the box chart below. Suppose we start with :X equal to one.

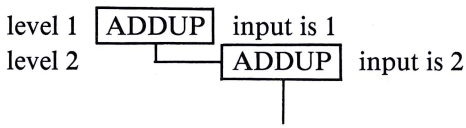


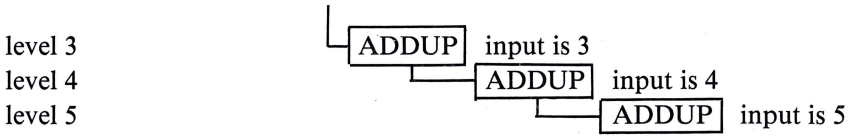
Notice that each procedure has one number as an input; this is very close to our situation. Is there a way to add together the inputs of each level of COUNTUP? That is, can we add level 1 COUNTUP to level 2 COUNTUP and so on (recursively)? Yes, if we use OUTPUTs. The way to pass data from one level to another level is to use the OUTPUT primitive.

Here is our result.

```
?TO ADDUP :X
>IF :X = 5 [OUTPUT 5]
>OUTPUT :X + ADDUP :X + 1
>END

?PRINT ADDUP 1
15
```

Hey, look at that. It works! Let's use the box chart again to explain this.





Consider ADDUP at level 3. Look at each line in the procedure. We'll call the title line, line 1. Here is the procedure again:

```
line 1 ..... TO ADDUP :X
line 2 ..... IF :X = 5 [OUTPUT 5]
line 3 ..... OUTPUT :X + ADDUP :X + 1
line 4 ..... END
```

And here is the analysis of level 3 of the procedure:

line 1 is ADDUP 3

line 2 is "False, so nothing is done

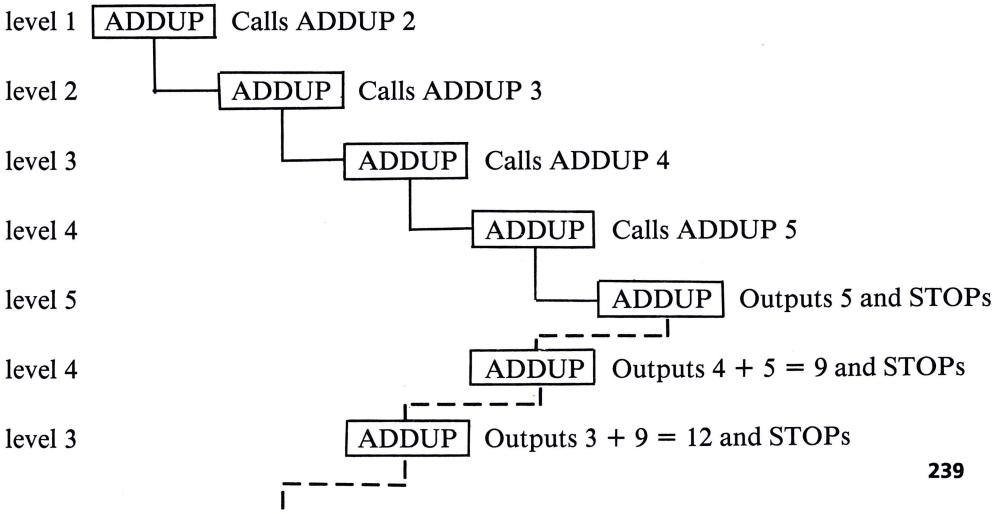
line 3 is the key. ADDUP 3 outputs the number 3 plus (+) whatever ADDUP 4 gives it.

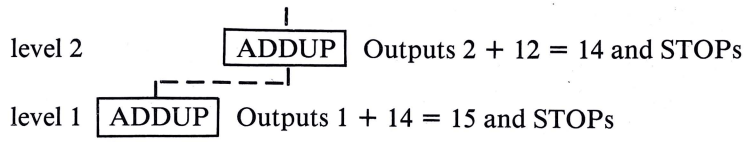
We see that ADDUP 3 adds together two numbers, its input, 3, and whatever ADDUP 4 outputs.

Let's look at ADDUP 4.

ADDUP 4 adds together the number 4 and whatever ADDUP 5 outputs. ADDUP 5 just outputs the number 5 so ADDUP 4 adds together 4 + 5 to give 9. ADDUP 4 outputs a 9 to ADDUP 3.

Let's look at the box chart again, this time from the point of view of outputs and the sequence of procedure calls.





Here we see the magic of recursion. Each procedure does a small part of the job and passes it on to the next level.

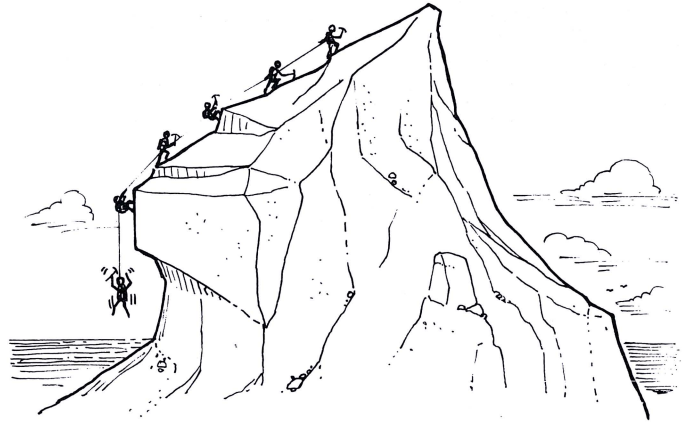


Fig. 9-3. OUTPUT is used to pass data between recursive levels.

There is a slightly better way to write this program. One problem is that to add up numbers to 40 or 100 would require editing. Let's change the input and IF condition as follows.

```
?TO ADDUP :X
>IF :X = 1 [OUTPUT 1]
>OUTPUT :X + ADDUP :X - 1
>END

?PR ADDUP 40
820

?PR ADDUP 100
5050
```

Wow! Look at how easy it is to get Logo to add up lots of numbers. We have reversed the sequence of numbers in the input so that we start at the top and decrease by one each time. An IF condition is always required in "recursion with outputs" to stop the lowest level procedure. Each OUTPUT statement stops each procedure in turn. No procedure actually runs

line 4 (END). All procedures end with line 3 (OUTPUT :X + ADDUP :X − 1). This is because OUTPUT stops the current procedure in order to pass a value to the procedure that called it.

Here are two variations of the ADDUP procedure.

```
?TO MULTUP :X
>IF :X = 1 [OP 1]
>OP :X * MULTUP :X − 1
>END
```

We just substituted multiplication (*) for addition (+). This is also known as a "factorial" program. A sample run produces

```
?MULTUP 5
120
```

In our next example, suppose we want to add up a list of numbers, not just 1, 2, 3, 4 . . . How would that work?

```
?TO SUMLIST :LIST
>IF BL :LIST = [ ] [OP LAST :LIST]
>OP (FIRST :LIST) + (SUMLIST BF :LIST)
>END

?SUMLIST [5 14 −6 2]
15
```

Notice that SUMLIST is almost identical to ADDUP. The only real differences are:

1. The IF condition tests for the LAST number in the list.
2. FIRST :LIST is substituted for :X
3. BF :LIST is substituted for :X − 1

The parentheses are necessary to give the proper inputs to plus (+).

Let's try one more example of using OUTPUTs with recursion. Here is a program that searches a list of numbers for a specific number and outputs that number if found. Our first attempt at writing the procedure doesn't quite work right. See if you can find the problem.

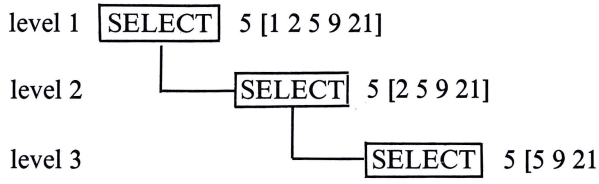
```
?TO SELECT :NUM :LIST
>IF (FIRST :LIST) = :NUM [OUTPUT :NUM]
>SELECT :NUM BUTFIRST :LIST
>END.
```

If we run the procedure with the following input

```
?SELECT 5 [1 2 5 9 2]
```

we get an error message (I DONT KNOW WHAT TO DO WITH 5 . . .)

The problem here can be seen with the box chart showing inputs:



We see that level 3 SELECT outputs the number 5, but to where?

The previous SELECT procedure, level 2 SELECT, gets the number 5 in line 2. But, this input is not fed to any primitive or procedure so Logo produces an error message.

What we want to happen is for level 2 SELECT to take the input and pass it on to level 1 SELECT. The solution to this problem is to use the primitive OUTPUT with the recursive call as follows.

```
?TO SELECT :NUM :LIST
>IF (FIRST :LIST) = :NUM [OUTPUT :NUM]
>OUTPUT SELECT :NUM BF :LIST          We added OUTPUT here
>END
```

Now, each lower level called passes the value :NUM up to the next higher level. The toplevel SELECT also outputs the :NUM. Thus, SELECT is an operation.

When we use recursion with OUTPUT, it's as though we've created a little colony of worker bees. Each level of recursion does one small part of the job and relies on the lower levels to continue the job. The primitive, OUTPUT, joins each level together so that the whole is more than just the sum of its parts. Recursion hums along busily outputting a honey of a result.

9.6 Summary

Here are the major points introduced in this chapter.

1. There are four ways to give inputs their values. These are
   a. Call the procedure with an input value specified as in STAR 30.
   b. Use the MAKE primitive to assign the value to the input before it is needed as in MAKE "SIZE 30
   c. Use READLIST and MAKE in an interactive program.
   d. For subprocedures, pass the input through each procedural level.
2. Inputs always create local variables.
3. The MAKE primitive always creates global variables with two exceptions.
   a. When changing a variable created by an input.
   b. When creating a variable localized by the LOCAL primitive.
4. Local variables are preferred when possible, to give procedures independence.
5. The primitive OUTPUT is used to create procedures that are operations.
6. Many utilities, like GETA.NUMBER and RANPICK use outputs to make them operations and useful building blocks.
7. OUTPUTS are often used in recursive procedures to pass data from one level to the next.

9.7 Review Questions and Exercises

Review Questions

1. Can a procedure contain an input in the title line even though the procedure doesn't use it?
2. MAKE takes two inputs. What kind of data objects are acceptable for the first input? The second input?
3. State two advantages of using inputs to using the MAKE primitive for assigning values to variables.
4. What is meant by the term "global variables"?
5. State two advantages of local variables to global variables.
6. How can you tell if you have any global variables in your workspace? How can you get rid of them if they are there?
7. Is READLIST an operation? If so, what does it output?
8. Why is it essential that the primitive OUTPUT stop the current run-

ning procedure?

9. Why do recursively defined operations have to end with the line

```
OUTPUT . . .
```

where . . . consists of the rest of the line?

10. Can you have a PRINT statement in an operation? Explain.

Exercises at the Terminal

1. Write the procedure AA, BB, CC and STAR as given in the text. Place a line with TYPE "AA TYPE CHAR 32 PRINT :SIZE (substitute BB for AA, etc) in each procedure. Then try each method of assigning STAR its input value.
2. Create an interactive HOUSE program that draws each component of the HOUSE with different size pieces possible.
3. Create an interactive thesaurus that lets you select which word to use instead of picking one randomly.
4. Take one of the operations like ADDUP or SELECT and add a PRINT statement in it. What happens? Why?
5. Create a procedure called COUNT.W that counts the number of letters in a word. It would do the same for words that COUNT does for lists, i.e., count the number of elements.

Answers to Review Questions

(1) Yes. (2) word only, any data object (3) Keeps variable local, lets you change the value easily when running the procedure (4) The variable is available to all procedures at all times (5) A procedure with local variables is more independent, less chance of bugs developing, not part of workspace (6) Use PONS, Use ERN "variable or ERNS to get rid of all global variables (7) Yes, it outputs whatever is typed at the keyboard before hitting a RETURN (8) Because OUTPUT passes a value on to the procedure that called it for that procedure to use. You couldn't have two procedures running at the same time. (9) Because each lower level procedure must pass the value (whatever value the operation is designed to OUTPUT) up to the next higher level. (10) Yes, as long as it is not used in place of OUTPUT.

CHAPTER 10

LOGO AND MATHEMATICS

CONCEPTS:

using numbers
arithmetic operations
drawing geometric shapes
using trig functions
graphing algebraic equations
calculating statistics

PRIMITIVES:

*+, −, *, 1, SUM, PRODUCT, QUOTIENT, REMAINDER, INT, ROUND, ARCTAN, SIN, COS, SQRT, >, <, =*

10.0 Introduction

"I know you like to draw, but can you do mathematics?" asked the Hare of the Tortoise.

"Sure!" replied the Tortoise, "I can do whatever you tell me."

"That sounds great," said the Hare, "Can you do Algebra, Trigonometry, Geometry and stuff like that?"

"Certainly," said the Tortoise, "Just tell me what to do."

"Well, I really like three-dimensional drawings. Can you draw shapes like cubes?" asked the Hare.

"Of course," said the Tortoise, "Just tell me exactly what you mean by a cube."

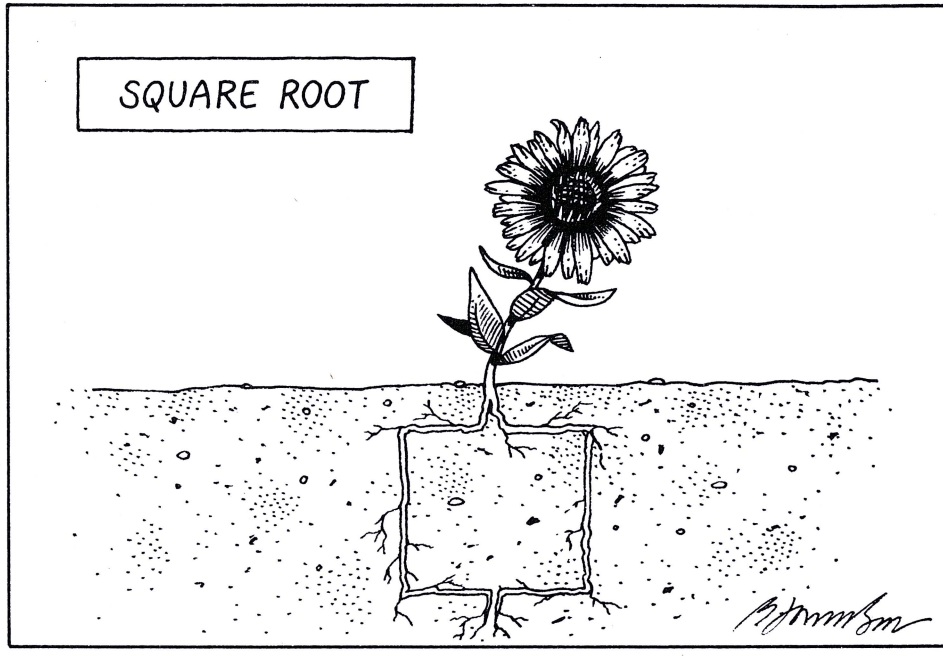
"You know," said the Hare, "A cube looks like a square with depth. It has three dimensions."

"I know how to draw a square," replied the Tortoise, "But you have to be a little clearer about 'depth.' Tell me exactly which direction to add the depth lines and how deep do these lines go?"

"Hmmm, do you really know how to draw cubes and do mathematics?" queried the Hare.

"I can do whatever you tell me to!" responded the Tortoise.

We'd better help these two out. In fact, Logo does have quite a bit of mathematics capability and the turtle certainly knows how to draw squares, circles, triangles and other shapes from our utility kit. But if we want to see "3-d" shapes like cubes and if we want to do mathematics, we will have to teach the turtle some new tricks.



Formal mathematics has often been a formidable topic for many people. However, in this chapter, we will explore "informal" mathematics, emphasizing those areas where turtle-graphics can be put to good use. We will find that graphics touches just about every branch of mathematics, including arithmetic, geometry, algebra, trigonometry and statistics. However, if you wish, you can skip most of this chapter for now.

Let's begin by seeing how Logo handles numbers.

10.1 Numbers

Numbers appear to be firmly embedded in nature. Atoms have specific numbers of electrons and protons. Specific numbers of atoms clump together to form people, planets and stars. Most living species have biological clocks that tick off the number of hours, days, lunar months or years that affect their well-being.

Although Logo uses a number system based on two internally, it provides us with the familiar base 10 system externally. In addition, Logo has incorporated a modern feature to extend the range of numbers we can easily use called "scientific notation."

The largest numbers Logo can play with are those written in scientific notation. Logo will accept numbers as large as 10^{38} and as small as 10^{-38}. The number 10^{38} stands for 1,000 . . . 000, that is, the number 1 with 38 zeros. More familiar numbers would be

`3*10^3` which is 3,000
`3*10^6` which is 3,000,000

Logo uses the symbol "E" to stand for 10 raised to a positive number (or "exponent") and the symbol N to stand for 10 raised to a negative number. Thus,

`3E3` is 3,000
`3E6` is 3,000,000
`3N3` is .003

The general form of scientific notation is shown in Fig. 10-1 below.

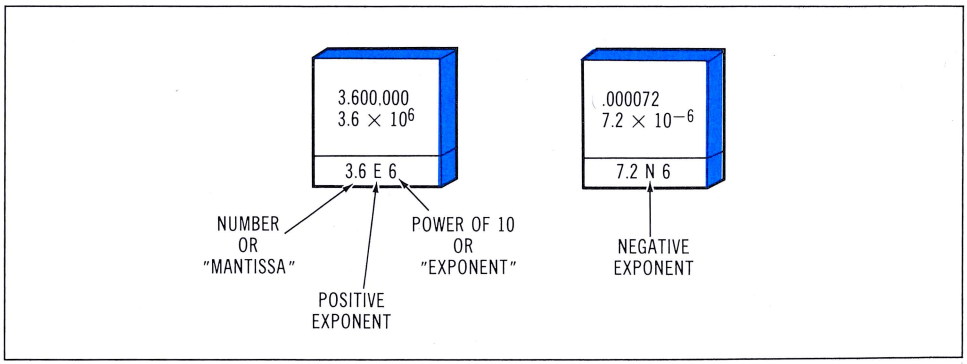


Fig. 10-1. Scientific notation with Logo.

Most computer languages use a similar representation for powers of ten since they cannot display or read superscripts or the "exponent" above the line.

Here are some examples of Logo numbers:

Table 10-1. Types of Logo Numbers

Integer Numbers	1	360	86400
Real or Decimal Numbers	3.14159	98.6	2.54
Positive Power Numbers	1E6	2.998E8	6.02E23
Negative Power Numbers	6.67N11	1.67N27	6.63N34
Negative Numbers	−273	−.001	−1.602N19

Logo can work with as many as seven digits in one number. We can have numbers like

1 1.2 1.234567 .1234567

but not

1.2345678.

If you enter a number with more than seven digits, Logo rounds it off. For example, 1234567.89 is converted to 1.234568E6.

Now that we see the kinds of numbers Logo likes, let's see how Logo combines numbers.

10.2 Arithmetic

We have already used the four basic arithmetic operators, +, −, *, / . These primitives are all operations that output their result to be used as input to another command or operation. Thus, we have expressions like

```
?PRINT 3*4
12
```

If we have a lot of mathematical calculations to do, we can make life a

little easier by turning Logo into a "calculator" as follows (Note: we will be leaving the > symbol out of our listings from now on):

```
TO CALCULATOR
PR RUN READLIST
PR[ ]
CALCULATOR
END
```

Here is a sample run.

```
?CALCULATOR
3 - 4
-1

6 * 7
42
```

The secret to this procedure is the line.

```
PR RUN READLIST
```

READLIST stops the program until a problem is typed in. Then, RUN runs the list with the arithmetic expression in it. This recursive program will continue until you stop it with a CTRL-G command.

The four operators can be placed in a more complex arithmetic expression. For example

```
?PR 4 * 3 + 8/(2 - 6)
10
```

The rules for handling these expressions, called the "rules of precedence," are the same rules we use in everyday life.

top priority..... carry out the operations within parentheses
next priority..... do multiplication and division
lowest priority ..... addition and subtraction

The minus sign (−) must be used with special care since it can take one input to produce a negative number (−4) or two inputs to produce subtraction (6 − 4). The location of the minus sign determines how Logo interprets it.

7 − 5 is 2
7− 5 is 2
7−5 is 2

However,

7 −5 is two numbers, a 7 and a −5.

Here is a procedure with a very common, but difficult to find, error.

```
TO COUNTDOWN :X
IF :X = 0 [STOP]
PR :X
COUNTDOWN :X −1
END
```

The recursive call should be written as

```
COUNTDOWN :X− 1
```

or

```
COUNTDOWN :X − 1
```

in order for subtraction to occur.

A simple rule of thumb is to always include a space after the minus sign if you want subtraction.

The four operations, +, −, *, /, are called "infix" operations, since the symbol is fixed "in between" the numbers.

Logo also provides addition, multiplication and division in "prefix" form where the operation "precedes" the numbers. Thus, we have expressions like

```
?PR SUM 5 6
11
```

One advantage to SUM and its prefix partner, PRODUCT, is that they can take multiple inputs when used with parentheses. For example,

```
(SUM 1 2 3 4 5 6)
  outputs 25
```

```
(PRODUCT 1 2 3 4 5 6)
  outputs 720
```

These results are the same as produced with the ADDUP and MULTIUP procedures from the last chapter. The procedures, however, have the advantage of easily taking large numbers of inputs for summing or factoring.

Other arithmetic primitives include SQRT, QUOTIENT, REMAINDER, INT, ROUND, >, <, =. For example, SQRT takes one input and outputs the square root.

```
?SQRT 4
2

?SQRT 200
14.1421
```

A nice procedure using SQRT is called DIST.HOME, which calculates the distance from the turtle's present location to the HOME position [0 0].

The procedure looks like this.

```
TO DIST.HOME
OP SQRT (XCOR * XCOR) + (YCOR * YCOR)
END
```

Here is a sample run.

```
?SETPOS [50 50]
```

the turtle moves to point X=50, Y=50

```
?PR DIST.HOME
70.7107
```

We can use the procedure DIST.HOME in an interesting program called RANWALK. It is also known as the "drunkard's walk" since the direction taken is chosen completely at random. (You can simulate the drunkard's walk by having someone blindfold you and spin you around a dozen times or so, then setting you "free." This won't be a true stagger unless you keep each "step" size you take the same value. We don't recommend this, especially if you're trying to program in Logo at the same time.)

Here is the program and a sample run.

```
TO RANWALK
CS
(LOCAL "STEP "N)
MAKE "STEP 10
PR [HOW MANY STEPS?]
MAKE "N FIRST RL
WALK :N
PR SE [THE ACTUAL DISTANCE HOME IS] DIST.HOME
PR SE [THE THEORETICAL DISTANCE IS] (SQRT :N) * :STEP
END

  TO WALK :N
  IF :N = 0 [STOP]
  RT RANDOM 359
  FD :STEP
  WALK :N - 1
  END
```

```
THE ACTUAL DISTANCE HOME IS 120.345
THE THEORETICAL DISTANCE HOME IS 133.267
```

The heart of the program is the line, RT RANDOM 359, in the procedure WALK, which turns the turtle to an angle between 0 and 359 degrees. Then the turtle goes forward 10 "steps" and WALKs again. After walking N times, the turtle stops and the program uses DIST.HOME to tell how far the turtle got.

The theoretical distance from home can be derived from trigonometry. We won't do it, since it's a bit tricky. We'll state the result, which is

Theoretical Distance = (SQRT N) * STEP :SIZE

The difference between the theoretical and actual distances home will vary considerably for one run. However, if you average the results for dozens of runs (or staggers), they should get quite close.

An interesting sidelight is that both results apply to the diffusion of gas molecules in air. For example, perfume molecules have a "step" size of about 10N6 feet. How many collisions are needed for these molecules to travel five feet?

Now let's look at another arithmetic primitive, REMAINDER.

The REMAINDER primitive may seem a little strange at first glance. It outputs the remainder obtained when dividing two numbers. For example,

```
REMAINDER 11 7
 outputs a 4


REMAINDER 25 4
 outputs a 1
```

This primitive can be used to tell if a number is divisible by 5 or any other number. For example,

```
TO DIVIDE5 :N
OUTPUT 0 = REMAINDER :N 5
END

?PR DIVIDE5 25
TRUE

?PR DIVIDE5 27
FALSE
```

The procedure, DIVIDE5, is handy to put into recursive loops to make some specific action occur every fifth loop. Here is a trivial example.

```
TO COUNT5 :X
IF DIVIDE5 :X [PR :X]
COUNT5 :X+1
END

?COUNT5 1
5
10
```

```
15
.
.
.
```

(use CTRL-G to stop this.)

In a later section on Algebra we use a similar technique to print the position of the turtle with ever tenth loop in plotting equations.

We'll summarize the discussion of arithmetic operators in Table 10-2.

Table 10-2. Logo's Arithmetic Primitives

Infix	Example	Outputs
+	5 + 3	8
−	5 − 3	2
*	5 * 3	15
/	5/3	1.66667
=	5=3	"False
<	5<3	"False
>	5>3	"True
Prefix		
SUM	SUM 5 3	8
	(SUM 5 3 2 4)	14
PRODUCT	PRODUCT 5 3	15
	(PRODUCT 5 3 2 4)	120
QUOTIENT	QUOTIENT 5 3	1
REMAINDER	REMAINDER 5 3	2
Other Primitives		
ROUND	ROUND 5.67	6
INT	INT 5.67	5
SQRT	SQRT 9	3

10.3 Geometry

From a geodesic dome to the Golden Gate Bridge to the pyramids, geometry has fascinated and shaped our thinking processes. Over 2000 years ago,

Euclid set out to create the rules that govern shapes on a flat surface. We still use Euclidean geometry today. Since then, geometry has divided into many branches including plane geometry, solid geometry, analytical geometry, Riemann geometry to name just a few and very recently, fractal geometry and turtle geometry.

The latter can be quite complex. For example, the book, "Turtle Geometry," by Abelson and DiSessa, MIT Press, 1981, was written to provide high school and college students with a rich environment for exploring two- and three-dimensional shapes with a Logo-like language. In this section, we will limit ourselves to two very basic ideas that produce interesting designs.

10.3.1 Two Points Define a Line

The shortest distance between two points is the straight line. Previously, we used the primitive FORWARD to draw straight lines. Another way to draw lines is to take the end points or X-Y coordinates of a line and use SETPOS to draw it. Here is one way to write the procedure:

```
TO LINE :PTA :PTB
PU SETPOS :PTA
PD SETPOS :PTB
END
```

Since SETPOS requires a list as input, PTA and PTB must be given in List form. For example,

```
?LINE [-100 -100] [-100 100]
```

Now this is nice, but not spectacular. However, we can create rather clever designs by using recursion and letting points A and B move. For example, let's start with point A and B shown below and move them in the direction of the arrows. We'll call the procedure ARCH1.

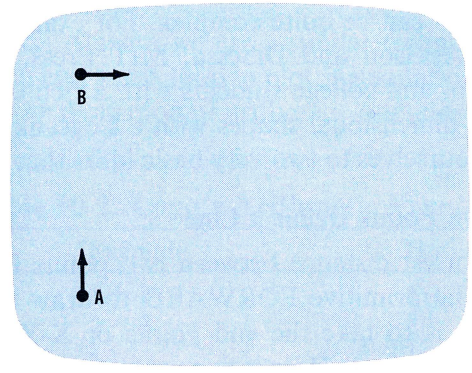
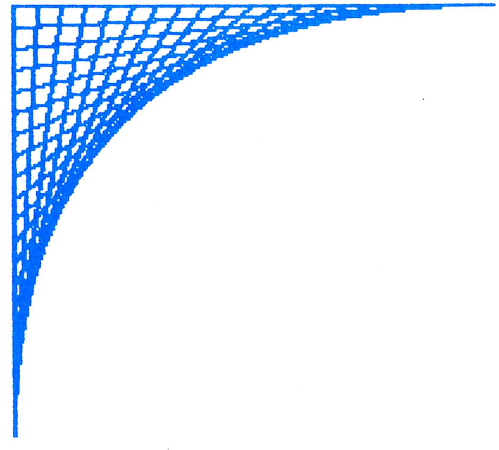


Fig. 10-2. Moving the end points, Point A and Point B.

```
TO ARCH1 :INC
IF :INC>200 [STOP]
MAKE "PTA SE -100 -100+:INC
MAKE "PTB SE -100+:INC 100
LINE :PTA :PTB
ARCH1 :INC+10
END
```

Here is a sample run.

```
?ARCH1 0
```



We called this procedure ARCH1 because we thought you might like to create ARCH2 that draws lines from point B to point C (which starts at X=100 Y=100) as shown in the figure below.

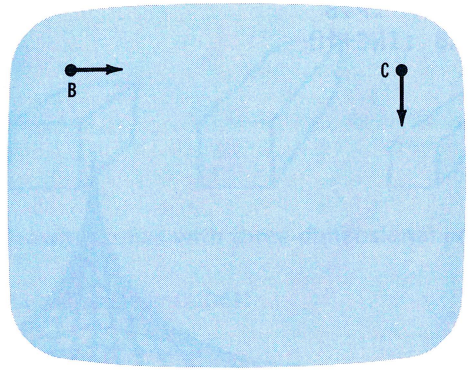


Fig. 10-3. Moving the end points, Point B and Point C.

Another version of this figure can be made by changing the starting points. The figure below is known as an "astroid." Here is the main program and its result:

```
TO ASTROID
SLIDEAB 0
SLIDEBC 0
SLIDECD 0
SLIDEDA 0
END
```

The starting points are shown in the figure below.

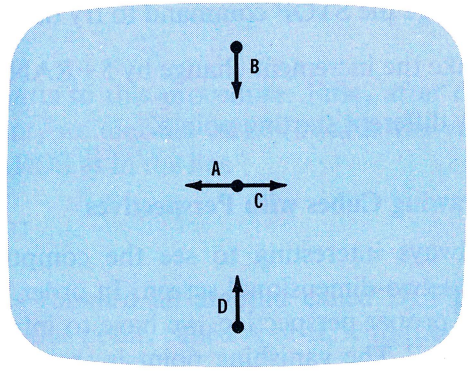
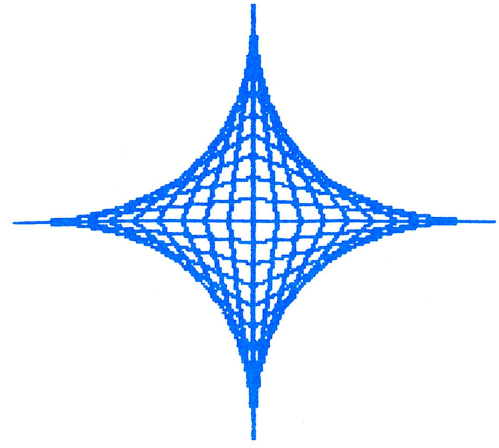


Fig. 10-4. Starting points for astroid.

```
TO SLIDEAB :INC
IF :INC>100 [STOP]
MAKE "PTA SE 0-:INC 0
MAKE "PTB SE 0 100-:INC
LINE :PTA :PTB
SLIDEAB :INC+10
END
```



Here, we have just shown one of the four subprocedures and leave the other three subprocedures as an exercise. There are other ways to experiment with this figure. These include:

1. Change the amount added to :INC.
2. Change the STOP command to try other values.
3. Make the increment change by 5+RANDOM 5 to see what happens.
4. Try different starting points.

10.3.2 Drawing Cubes with Perspectives

It is always interesting to see the computer draw a three-dimensional object on a two-dimensional screen. In order to draw a shape, a cube in our case, with proper perspective, we have to introduce a point called the "vanishing point." The vanishing point is that point far away where an object has zero size. It is easier to see than to explain. Here are four different

cubes shown with all of the "depth" lines heading towards the vanishing point.

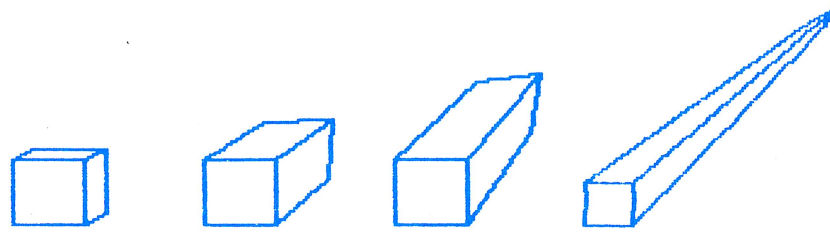


Fig. 10-5. Drawing cubes with three-dimensional perspective.

And the procedure that draws the cubes:

```
TO PCUBE :SIZE :DEPTH
WINDOW
MAKE "PT SE 6 * :SIZE 6 * :SIZE
REPEAT 4 [FD :SIZE RT 90]
FD :SIZE
SETH TOWARDS :PT
FD :DEPTH MAKE "A POS BK :DEPTH
SETH 90 FD :SIZE
SETH TOWARDS :PT
FD :DEPTH MAKE "B POS BK :DEPTH
SETH 180 FD :SIZE
SETH TOWARDS :PT
FD :DEPTH
SETPOS :B
SETPOS :A
HT
END
```

There are two key parts to this procedure. First, after defining the location of the vanishing point, we aim the turtle towards the vanishing point using the primitive TOWARDS as in the line

```
SETH TOWARDS :PT
```

Then, the turtle goes forward an amount called DEPTH. Now comes the second key. We mark this point, called A or B, with the command

```
MAKE "A POS
```

We do this so that we can connect up the three depth lines at the end of the procedure as shown below.

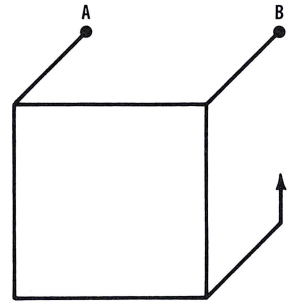


Fig. 10-6. Connecting the depth lines.

Here are some suggestions for further exploring.

1. Try moving the vanishing point.
2. Try to make 3-dimensional rectangle or triangles.
3. Draw two or three cubes each with its own vanishing point.
4. Make a 3-dimensional rectangle into a road.
5. Add 3-dimensional shapes alongside a 3-dimensional road.
6. Going back to the cube, can you add the "hidden lines" as dotted lines. The hidden lines are the other three sides of the cube.

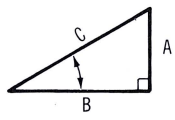
10.4 Trigonometry

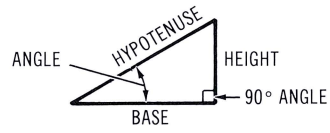
Trigonometry is the branch of geometry that deals with triangles. Surveyors use "trig" to measure the size of trees, buildings and plots of land. Engineers and scientists use trig to describe how objects vibrate and how waves travel, among other things. We will use trig to draw triangles. We are going to focus on the simplest kind of triangle, the "right" triangle. Logo provides three primitives for use with right triangles. These primitives are SIN, COS, and ARCTAN.

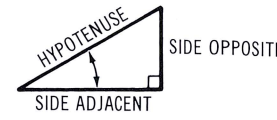
10.4.1 The Right Triangle

The right triangle is called "right" because it contains a "right" angle. (A big help that is!) A right angle is 90 degrees. (If you make a "sharp

right'' turn, you turn 90 degrees.) The relationship between the sides, angles and Logo primitives for the right triangle are shown below.







$$\text{SIN (ANGLE)} = \frac{A}{C} = \frac{\text{HEIGHT}}{\text{HYPOTENUSE}} = \frac{\text{SIDE OPPOSITE}}{\text{HYPOTENUSE}}$$

$$\text{COS (ANGLE)} = \frac{B}{C} = \frac{\text{BASE}}{\text{HYPOTENUSE}} = \frac{\text{SIDE ADJACENT}}{\text{HYPOTENUSE}}$$

$$\text{ARCTAN} \left(\frac{A}{B}\right) = \text{ANGLE}$$

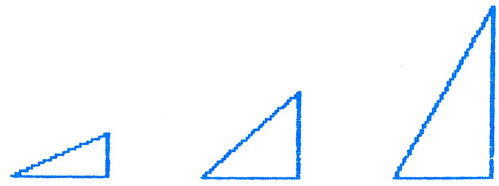
Fig. 10-7. The right triangle and Logo primitives.

Although all three definitions for the SIN are correct, the last one is preferred since it is most general. (The last definition works for all angles, not just the one made with the horizontal axis).

However, the easiest figure for us to use is the middle one. Let's use it to create a procedure that draws right triangles when given the base and the height.

```
TO RTRI :BASE :HT
LOCAL "POS
MAKE "POS POS
RT 90
FD :BASE
LT 90 FD :HT
SETPOS :POS
PR SE [THE BASE IS] :BASE
PR SE [THE HEIGHT IS] :HT
PR (SE [THE LOWER ANGLE IS] ARCTAN :HT / :BASE [DEGREES])
END
```

Here are three sample runs:



This procedure uses a trick we learned in the last section on vanishing points. The secret is to mark the starting position with the command

```
MAKE "POS POS
```

POS is a Logo primitive that outputs the turtle's current XY-coordinates. We also create a variable of the same name (whose value is :POS, which is different from POS). SETPOS is then used to draw the hypotenuse.

Here are three variations of this procedure you might try.

1. Drawing a right triangle given the angle and the base, or given the base and the hypotenuse.
2. Put RTRI in a recursive procedure that draws triangles of increasing heights.
3. Combine (2) and (3) to draw a family of triangles all having the same hypotenuse, but different angles. (What does this begin to look like?)

10.4.2 Using the SIN and COS Values

Although trig was created around the triangle, the way that SIN and COS change for different angles is of wider interest. Let's look at how the SIN of an angle changes as the angle changes. We'll do this three ways; first numerically, then using two figures.

When the angle is zero, the height of the triangle is zero, so that

SIN (0 degrees) = height/hypotenuse = 0

When the angle is 90 degrees, the base is zero and the height just equals the hypotenuse. So

SIN (90 degrees) = height/hypotenuse = 1

In order to see what happens in between these two extreme angles, here is a simple procedure.

```
TO PRSIN :ANG
PR SIN :ANG
PRSIN :ANG +10
END
```

```
?PRSIN 0
0
0.173648
0.34202
0.5
0.642788
0.766044
0.866025
0.939693
0.984808
1
0.984808
0.939693
AND SO ON
```

A better way to "see" the values of SIN is to graph them. Here are two different representations.

Fig. 10-8. Two graphs showing how SIN changes.

The first graph plots the value of the SIN along the Y-axis and the angle itself along the X-axis starting at point (0,0). Here is the procedure that drew it.

```
TO XSIN :ANG
SETPOS SE :ANG 30*SIN :ANG*5
XSIN :ANG+5
END
```

The resulting curve is often called a "sine wave" and is seen in lots of places (for example, in the ripples on a pond).

The second graph plots the value of SIN as the radial distance from the home position [0 0] as we rotate the turtle through 360 degrees. The turtle goes FORWARD, then BACK, then rotates 2 degrees and goes FORWARD again. Each FORWARD step depends on the value of the sine. The turtle is

brought back, rotated again for another step. Notice that the "lobe" is a maximum at an angle of 90 degrees, which is what we expect for the SIN function. This graph also demonstrates how to plot curves using "polar" coordinates. You might also expect a lobe at 270 degrees. It's there, but since the SIN 270 is minus one (−1), the turtle retraces the same lobe as at 90 degrees. You can use the utility ABS (for absolute value) introduced at the end of this chapter to change the radius and see both lobes.

Here is the procedure that drew the graph.

```
TO ROTSIN :ANG
MAKE "RADIUS 100*SIN :ANG
FD :RADIUS
BK :RADIUS
RT 2
ROTSIN :ANG+2
END
```

Some ideas for you to explore include:

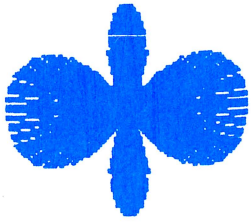
1. Try the same procedures using the COS function.
2. What happens if you multiply the angle (:ANG) in the SIN or COS function by 2 or 3 or 4? Try it, you'll like it!
3. Add a "stop" statement to stop the procedure at some angle, say 180 degrees.
4. Try some other equations for the radius in the "polar" equation like

```
MAKE "RADIUS 40 + 40 * COS 2 * :ANG
MAKE "RADIUS (30 * COS 5 * :ANG) * ( 5 * SIN :ANG)
```

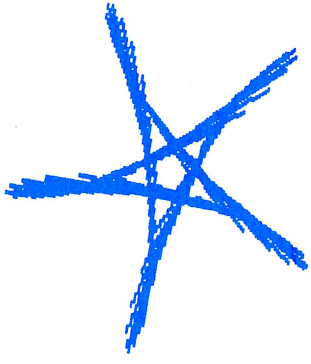
Here are two discoveries that we made while playing around:

```
TO SPURS :ANG
IF :ANG > 360 [STOP]
MAKE "RADIUS 50 * COS :ANG *2 + 50 * SIN :ANG *2
FD :RADIUS
BK :RADIUS
RT 2
SPURS :ANG + 2
END
```

And the result:



In the next example, we accidentally came back further than we went forward to get the following figure:



The procedure listing:

```
TO STARCOS :ANG
IF :ANG > 180 [STOP]
FD 50 * COS :ANG * 5
BK 63 * COS :ANG * 5
RT 3
STARCOS :ANG + 3
END
```

10.4.3 Combining Geometry with Trigonometry

The "cardioid" is a stimulating design found in many geometry books. It can be created by using a unique characteristic of the Logo turtle. If you look closely at Fig. 10-9, you'll see that it is made up from circles of increasing radius, then decreasing radius.

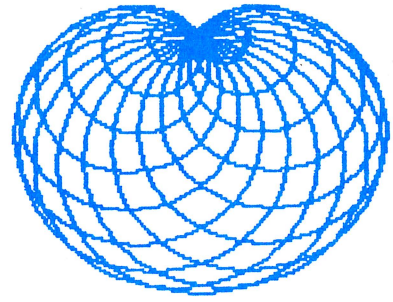


Fig. 10-9. The cardioid.

The largest circle is the one drawn with the turtle starting out facing the right (90 degrees). This means we can use the SIN function to change the circle's radius. Here is the procedure

```
TO CARDIOID
MAKE "ANG 10
MAKE "RADIUS 60 * SIN :ANG
DOCIRCLE :ANG :RADIUS
END

  TO DOCIRCLE :ANG :RADIUS
  IF :ANG > 180 [STOP]
  CIRCLE :RADIUS
  RT 10
  DOCIRCLE :ANG + 10 60 * SIN :ANG
  END
```

The secret to the program lies in the input to DOCIRCLE which is :ANG. The recursive call increases the angle, then we calculate a new radius using SIN :ANG, and then use the new radius to draw a new circle (drawn with the CIRCLE utility).

10.5 Algebra

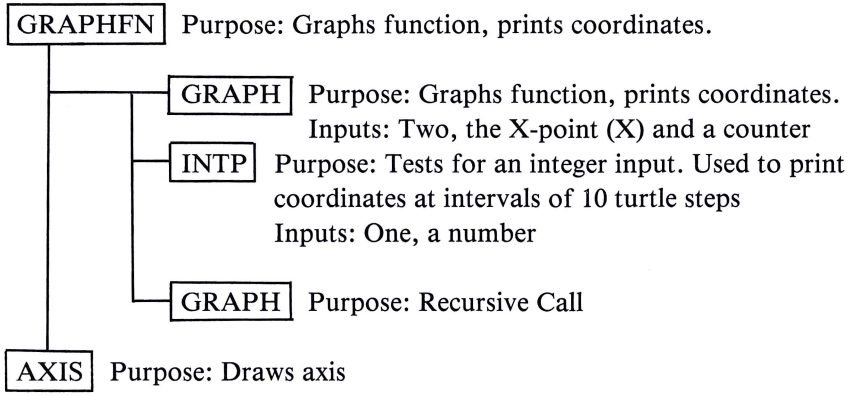
We have been using algebra throughout this book, ever since we introduced inputs as variables in Chapter 3. However, in this section we will look at two traditional algebra problems. The first problem is, "Can we graph the following function?"

```
3X - 2Y = 25
```

Of course we can! In order to do this, we first have to prepare the function, rewriting it so that Y is alone on the left side of the equation. The function now reads

```
Y = −25/2 + 3X/2
```

It is now ready for graphing. Here is a box chart showing the graphing program.


GRAPHFN
Purpose: Graphs function, prints coordinates.
GRAPH
Purpose: Graphs function, prints coordinates.
Inputs: Two, the X-point (X) and a counter
INTP
Purpose: Tests for an integer input. Used to print coordinates at intervals of 10 turtle steps
Inputs: One, a number
GRAPH
Purpose: Recursive Call
AXIS
Purpose: Draws axis

In the procedure, GRAPH, a "counter" is used to print the coordinates after every 10 points plotted on the graph. We do this to make the graphing process go faster.

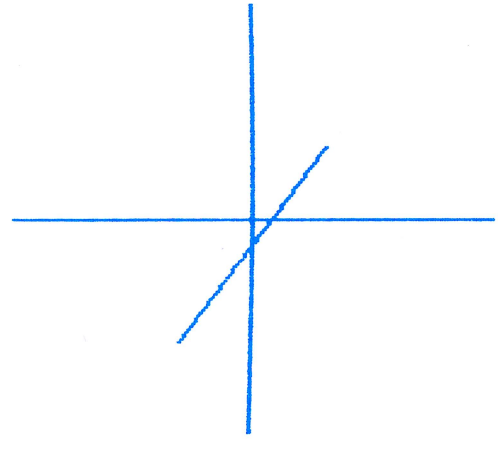
Here is a listing of the procedures:

```
TO GRAPHFN
PR [ENTER FUNCTION]
MAKE "FN RL
MAKE "X −30
MAKE "INC 1
MAKE "XSTOP 30
MAKE "Y RUN :FN
PU SETPOS SE :X :Y PD
GRAPH :X 1
AXIS
END

TO GRAPH :X :COUNTER
IF XCOR > :XSTOP [STOP]
MAKE "Y RUN :FN
```

```
SETPOS SE :X :Y
IF INTP :COUNTER / 10 [PR (SE [X =] :X CHAR 32 [AND Y =]
:Y)
WAIT 120]
GRAPH :X + :INC :COUNTER + 1
END

  TO INTP :N
  OP :N = INT :N
  END

TO AXIS
PU HOME PD
FD 100 BK 200 FD 100
RT 90
FD 100 BK 200 FD 100
HT
END
```

And the sample run:

```
?GRAPHFN
PLEASE ENTER FUNCTION
-25/2 + 3 * :X/2
```



The XY coordinates appear at the bottom of the screen.

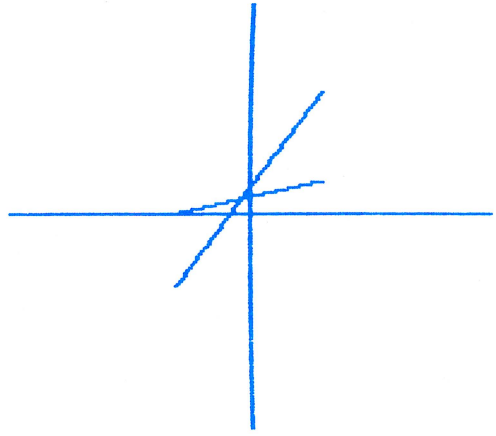
Let's leave this function on the screen and turn to the second Algebra

problem. "Can we solve two sets of equations with two unknowns?" For example, suppose we have the two equations

```
3X - 2Y = 25
-X + 4Y = -32
```

What is the solution? Can we get the turtle to solve this?

Yes! Just call the procedure, GRAPHFN, again, and enter the new equation. Here is the result.



The solution to these two equations is the point where they meet. We can look at the screen and estimate that the coordinates where the two curves cross is

```
X = -1
Y = -8
```

You may have to run the equations again to "estimate" the point where they cross more carefully. Also, you can change the time periods used for printing coordinates to improve this estimate.

Can this program be used to plot more complicated functions, like XE3 + 2XE2 −3X + 1? Sure. The only difficulty might be that the value of Y is too big and X too small for plotting on the screen. You can "scale" down the size of Y in the line of code in GRAPH that reads

```
SETPOS SE :X :Y
```

Just divide Y by 10 or 20 (:Y/10) or divide Y by a variable called

:YSCALE. Similarly or conversely (depending on circumstances and starting points), you can scale X up.

Now that we know how to solve Algebra problems, let's move on to Statistics.

10.6 Math Utilities in Statistics

We'll conclude this chapter on Logo and mathematics by returning to the topic we started with: Numbers. Now, however, we would like to create some math utilities that are useful in dealing with numbers. These utilities include:

Math Utilities

`MAX :A :B`	Outputs the maximum of two numbers
`MIN :A :B`	Outputs the minimum of two numbers
`HI :DATA`	Outputs the highest of a list of numbers
`LO :DATA`	Outputs the lowest of a list of numbers
`AVERAGE :DATA`	Outputs the average of a list of numbers

These utilities provide some elementary statistics of numbers.

Let's first look at the two utilities MAX and MIN.

```
TO MAX :A :B
IF :A > :B [OP: A]
OP :B
END

?MAX 1 2
 outputs 2


?MAX 10 10.1
 outputs 10.1


?MAX -5.0 -5.6
 outputs -5.0
```

The procedure MIN is defined similarly

```
TO MIN :A :B
```

```
IF :A>:B [OP :B]
OP :A
END
```

Now that we have a utility to find the MAX and MIN of two numbers, we can use recursion to do the same thing for a list of numbers.

Here is the HI procedure.

```
TO HI :DATA
IF (BF :DATA) = [ ] [OUTPUT FIRST :DATA]
OUTPUT MAX (FIRST :DATA) (HI BF :DATA)
END

?HI [1 2 7 - 4 3.1]
 outputs 7
```

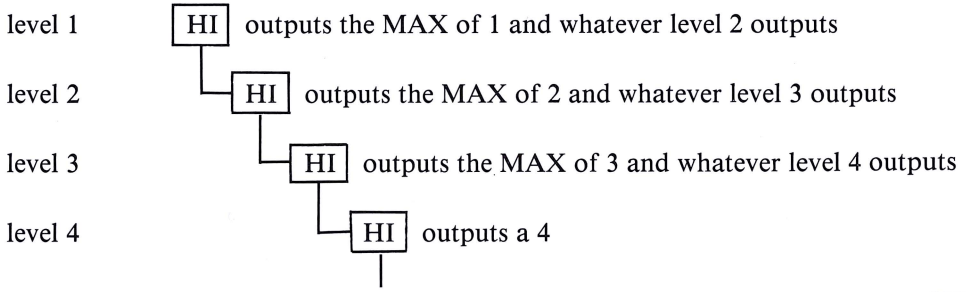
The HI procedure uses the concepts discussed in the last chapter—recursion and outputs—to create a series of subprocedures each one doing a small part of the job (using MAX). Let's review in detail how this works.

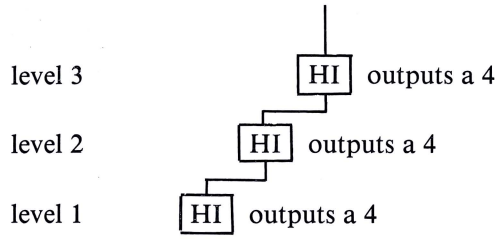
Here is the example

```
?HI [1 2 3 4 ]
 outputs 4
```

We'll use the box chart to explain what happens at each recursive level. Here is the HI procedure once more.

```
TO HI :DATA
IF (BF :DATA) = [ ] [OUTPUT FIRST :DATA]
OUTPUT MAX (FIRST :DATA) (HI BF :DATA)
END

?HI [1 2 3 4]
```





Thus each level of HI compares two numbers and outputs the maximum number. One more reminder. No procedure of subprocedure ever runs line 4 of HI since the primitive OUTPUT stops the current procedure whenever it is executed.

The procedure LO works just like HI:

```
TO LO :DATA
IF (BF :DATA) = [ ] [OUTPUT FIRST :DATA]
OUTPUT MIN (FIRST :DATA) (LO BF :DATA)
END

?PR LOW [-5 2 -5.1 3 -3 2E4]
-5.1
```

Our last math utility is AVERAGE. It uses the math utility developed in the last chapter called SUMLIST. SUMLIST takes a list of numbers as input and outputs their sum. Here are the two procedures.

```
TO AVERAGE :DATA
OUTPUT (SUMLIST :DATA)/COUNT :DATA
END

TO SUMLIST :DATA
IF (BL :DATA) = [ ] [OUTPUT FIRST :DATA]
OUTPUT (FIRST :DATA) + (SUMLIST BF :DATA)
END

?PR AVERAGE [5 6 7 8 9 10]
7.5

?PR AVERAGE [1 2 3 -1 -2 -3]
0
```

AVERAGE works by using the Logo primitive COUNT to count the number of numbers in the data list. Since COUNT is an operation that out-

puts a number, we can divide the result of SUMLIST by the result of COUNT. AVERAGE provides a vivid example of the value of operations.

Statistics is the branch of mathematics that deals with numbers that either come from the measurement of some quantity or come by chance. There is considerably more to statistics and of course, to all of mathematics than we have touched on in this chapter.

However, we wanted to demonstrate that Logo and the turtle can be called upon to give us new insights into the stimulating world of mathematics.

10.7 Summary

Here are the important points in this chapter.

1. Logo provides us with positive, negative and decimal numbers, like 1, 3, 17, −68, 9.4.
2. Numbers can be expressed in scientific notation where 2.0E6 is two million and 2.0N3 is .002.
3. We can use either infix operators like + or prefix operators like SUM to do arithmetic.
4. Logo has built-in rules, called the rules of precedence, that give top priority to parentheses and next priority to multiplication and division when these appear in complex expressions. Thus, 6 + 4/2 = 8 and (6 + 4)/2 = 5.
5. Logo has three primitives that are used with right triangles. These are SIN, COS, and ARCTAN.
6. The turtle can be used to draw figures that illustrate basic concepts in geometry, trigonometry and algebra.
7. All mathematic primitives in Logo are operations.
8. We created several math utilities to add to our utility library. These include MAX, MIN, HI, LO, and AVERAGE.

10.8 Review Questions

1. Write the following Logo numbers in scientific notation.

2.34E5
3.14149E0
7.5N3
−7.6N3

2. How would the following expressions be evaluated?
   PR 6 + (5*3)/2
   PR (10+10)/(2+2)+8
   PR 6−4/2
   PR 8 −1
   PR REMAINDER 11 2
3. Which of the following expressions is incorrect?
   SETPOS SE 100 50
   SETPOS 55 77
   SETPOS [33 44]
   SETPOS LIST 33 44
4. What kind of input does SIN take?
5. What would the following procedure produce?
   TO MYSTERY :A
   SETPOS SE :A 30*SIN 3*:A
   HOME
   RT :A
   MYSTERY :A+3
   END
6. Here is the procedure INTP.
   TO INTP :N
   OP :N = INT :N
   END
   What does it output?
7. What would the following lines of code produce?
   PR SUM MAX 3 2 MIN 5 4
   PR HI (SE MIN 5 3 MIN 4 2 MIN 3 1)

Answers to Review Questions

(1) 2.34×10^{5} 3.14159×10^{0} 7.5×10^{-3} -7.6×10^{3} (2) 13.5, 13, 4, error, 1 (3) SETPOS 55 77 (4) number (5) polar plot of sine (6) "TRUE or "FALSE (7) 7, 3

CHAPTER 11

MAKING DECISIONS WITH CONDITIONALS

CONCEPTS:

making choices
predicates: "TRUE and "FALSE
logical operations

PRIMITIVES:

IF, TEST, IFTRUE, IFFALSE, AND, OR, NOT, MEMBERP, WORDP, LISTP, NUMBERP, EMPTYP, KEYP, BUTTONP, DEFINEP, NAMEP, EQUALP, =, >, <

11.0 Introduction

When someone or something comes up to an automatic door at a supermarket, it opens, even if the would-be customer is a gun-flourishing masked man or a ravenous grizzly bear. The door has only one response, so right or wrong, it opens. It lacks options. Giving modern devices the ability to make decisions would make life more interesting. For example, your chair could grunt and groan if you gained weight beyond a certain point. Your bicycle could suggest you take a break when your pulse rate gets too high. Your tennis racquet could tell you to hit the ball closer to the racquet throat if you are missing its sweetspot. The parking meter could return your nickel and tell you the meter maid won't be by for another 2 hours. But chairs,

bicycles, racquets, and parking meters don't make decisions, so none of this happens (yet!).

Of course, there are devices that do make decisions. Cameras will set the proper exposure for you and tell you if you should use flash. Some automobiles tell you that they are getting low on gasoline. Others sample the exhaust gases and adjust their air-gasoline mixture accordingly. Some tape recorders set the recording level automatically. These devices all sense the environment, then make decisions based on what they find. We would like to bring decision-making of this level to our Logo programs.

Some computer programs are written the same way the automatic door behaves: without options. When something unexpected or inappropriate comes along, the program crashes or does something foolish. The key to making programs more intelligent is to give them a way to make decisions, to choose options. In Logo this is done with *conditionals.* We have already used one form of conditional, the IF primitive. In this chapter we will look at IF and other Logo conditionals and see how to use them to make intelligent, flexible programs. Let's begin with IF.

11.1 IF: Simple Form with Two Inputs

If it's raining, take an umbrella. If you have time, pick up a loaf of sourdough bread. If the voltmeter reads under 1.45 volts, replace the battery. If the tail is not wagging, don't pet the dog. Our lives are full of "if" decisions. The IF primitive gives Logo programs some of the same capabilities. Let's work through a few simple examples.

Here's one showing the simplest use of IF:

Listing

```
TO GREET
LOCAL "ANS
PR [WHO'S THERE?]
MAKE :ANS RL
IF "ANS = [SANTA CLAUS] [PR [FAR OUT!]]
PR SE "HOWDY, :ANS
PR [MAKE YOURSELF COMFY]
END
```

Run 1

```
?GREET
```

```
WHO'S THERE?
GOMER HICKS
HOWDY, GOMER HICKS
MAKE YOURSELF COMFY
?
```

Run 2

```
?GREET
WHO'S THERE?
SANTA CLAUS
FAR OUT!
HOWDY, SANTA CLAUS
MAKE YOURSELF COMFY
?
```

This simple form of IF has two inputs. The first input should be what Logo calls a *predicate.* This is simply something with the value "TRUE or "FALSE. (We'll see more about predicates shortly.) The second input, enclosed in braces, is an action to be taken if the predicate is true. Once the IF statement is processed, the procedure normally moves on to the next line. Thus, the "HOWDY" line gets printed whether or not Santa Claus is there. The exception to this rule occurs when the action in the IF statement decrees otherwise, for instance, if the action is a STOP command.

Using IF with STOP is very handy for stopping recursion loops. Here is a simple example of that.

Listing

```
TO GUESS.ME
LOCAL "ANS
PR [TO STOP ME, GUESS MY NUMBER]
MAKE "ANS RL
IF FIRST :ANS = 4 [STOP]
PR [NOPE, GUESS AGAIN]
GUESS.ME
END
```

Run

```
?GUESS.ME
TO STOP ME, GUESS MY NUMBER
```

```
3
NOPE, GUESS AGAIN
TO STOP ME, GUESS MY NUMBER
42
NOPE, GUESS AGAIN
TO STOP ME, GUESS MY NUMBER
4
?
```

We can include more than one action within the brackets. Here is a slight modification of GUESS.ME:

Listing

```
TO GUESS.ME1
LOCAL "ANS
PR [TO STOP ME, GUESS MY NUMBER]
MAKE "ANS RL
IF FIRST :ANS = 4 [PR [THAT'S IT! BYE] STOP]
PR [NOPE, GUESS AGAIN]
GUESS.ME1
END
```

Run

```
?GUESS.ME1
TO STOP ME, GUESS MY NUMBER
2
NOPE, GUESS AGAIN
TO STOP ME, GUESS MY NUMBER
4
THAT'S IT! BYE
?
```

If you have several actions you want done, you can lump those actions into a separate procedure and use the procedure name as the action:

Listing

```
TO GUESS.ME2
LOCAL "ANS
PR [TO STOP ME, GUESS MY NUMBER]
```

```
MAKE "ANS RL
IF FIRST :ANS = 4 [RESPOND STOP]
PR [NOPE, GUESS AGAIN]
GUESS.ME2
END

TO RESPOND
PR [OH WOW! YOU GOT IT! THAT'S JUST]
PR [TERRIFIC! I'M REALLY IMPRESSED!]
PR [I'M OVERCOME! BYE!]
END
```

We'll let you check to see if this works as expected. By the way, what do you suppose would happen if we moved STOP from GUESS.ME2 to RESPOND? Try it out, and make sure you understand why it works (or fails to work) as it does.

Actually, IF is not limited to just two inputs, but before we follow up on that, let's take a quick look at Logo predicates. Perhaps the most common examples are comparisons:

```
IF :NUMBER = 6 [STOP]
IF :COUNT > 7 [PR [THAT'S A LOT OF HAMBURGERS!]]
IF :AGE < 18 [PR [SORRY, YOU'RE TOO YOUNG]]
```

The predicate in the first example compares for equality. If :NUMBER and 6 have the same value, then the expression

```
:NUMBER = 6
```

has the value "TRUE; otherwise the expression has the value "FALSE. To check this, try the following:

```
?MAKE "NUMBER 6
?PR: NUMBER = 6
TRUE
?MAKE "NUMBER 8
?PR :NUMBER = 6
FALSE
?
```

Note that in Logo, MAKE is used to make :NUMBER equal to 6, and that = only checks to see if :NUMBER and 6 are equal.

Similarly, the expression

```
:COUNT > 7
```

has the value "TRUE only if :COUNT is greater than 7, and

```
:AGE < 18
```

has the value "TRUE only if :AGE is less than 18. (The > is the symbol for "greater than," and the < is the symbol for "less than.")

In summary, the first input for IF is something with the value "TRUE or "FALSE. It could be a Logo predicate, like those above or those we will discuss later. Or it could be a procedure that outputs "TRUE or "FALSE. Or it even could be the words "TRUE or "FALSE themselves, although that is not too useful:

```
?IF "TRUE [PR [OH YEAH]]
OH YEAH
?
```

The second input for IF is an action or series of actions contained within brackets. These actions can be primitives or procedures of your own.

The third input for IF . . . well, that is the topic of our next section.

11.2 IF: Complete Form with Three Inputs

Often we want the program to do one thing if a condition is true and a different thing if the condition is false. In Logo we can feed a third input to an IF primitive; this third input is a second list of actions (also enclosed in brackets) that get done if the condition is false. Here is an example:

Listing

```
TO QUERY
PR [DO YOU LIKE COMPUTERS? Y OR N?]
IF (FIRST FIRST RL) = "Y [PR [THANKS!]] [PR [SNIFF,]
PR [I LIKE PEOPLE]
END
```

Run 1

```
?QUERY
```

```
DO YOU LIKE COMPUTERS? Y OR N?
Y
THANKS!
I LIKE PEOPLE
?
```

Run 2

```
?QUERY
DO YOU LIKE COMPUTERS? Y OR N?
N
SNIFF,
I LIKE PEOPLE
?
```

When the predicate expression was true, the action in the first set of brackets was executed. When it was false, the action in the second set of brackets was executed. Then, as with the simple IF, the flow moved on to the next statement in the procedure.

The third input to IF, like the second, can be a single action or a series of actions. The actions can be Logo primitives or your own procedures.

What if we just want something done when the first input is false? You might guess, just leave out the second input and just have the first and third inputs. But how will Logo know that one input is missing? One solution is to have an empty input:

```
IF :AGE = 39 [ ] [PR [YOU'RE NOT JACK BENNY]]
```

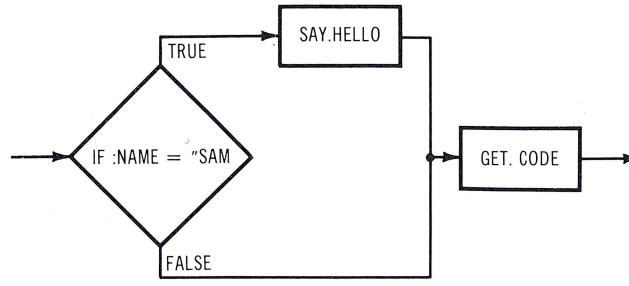
But that can be a bit confusing to the casual reader. A better approach is to reverse the condition by using the Boolean primitive NOT. ("Boolean" is derived from the name of George Boole, who worked on the mathematics of logic.) Thus we can say

```
IF NOT :AGE = 39 [PR [YOU'RE NOT JACK BENNY]]
```

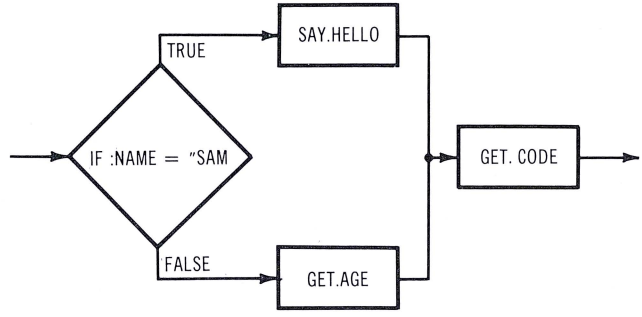
You can read this, "If age is not 39, print 'you are not Jack Benny.' " If something is "TRUE, then NOT something is "FALSE:

```
?PR NOT "TRUE
FALSE
?
```

Fig. 11-1 illustrates the difference in program flow for the two forms of IF.



1. IF :NAME = "SAM [SAY.HELLO]
   GET.CODE



2. IF :NAME = "SAM [SAY.HELLO] [GET.AGE]
   GET.CODE

Fig. 11-1. IF with two inputs vs. IF with three inputs.

Logo has three Boolean (or logical) primitives: AND, OR, and NOT. They can be used to combine and modify other expressions, much expanding the capabilities of IF. We'll look at them after we give this formal definition of IF.

IF	**(takes two or three inputs)**
Purpose	Take a given action (second input) if a condition (first input) is true. Optionally take another action (third input, if present) if the condition is false
Abbrev	None
Inputs	First input is any Logo data object or output having the value "TRUE or "FALSE. The second input is an action or

	series of actions enclosed in braces. The third input, if present, has the same form as the second
Outputs	None

Examples:

```
?IF 7 > 3 [PR [I THOUGHT SO]]
I THOUGHT SO
?IF 6 > 8 [PR [I THOUGHT SO]] [PR [I'M POOR WITH NUMBERS]]
I'M POOR WITH NUMBERS
?
```

11.3 The Primitives AND, OR, and NOT

Suppose you want a procedure to do something only if two conditions are met at the same time. For example, suppose you want to print the message "GO TO WINDOW 4" only if someone's age is greater than 18 and less than 29. The AND primitive lets you combine these two conditions together:

```
IF AND :AGE > 18 :AGE < 29 [PR [GO TO WINDOW 4]]
```

Then the message gets printed only if both relations following the AND are true. Note that AND takes two inputs, each of which should be something with a true or false value.

These examples may make the workings of AND clearer:

```
?PR AND "TRUE "TRUE
TRUE
?PR AND "TRUE "FALSE
FALSE
?PR AND "FALSE "FALSE
FALSE
?PR (AND 4=6 5=6 6=6 7=6 8=6)
FALSE
```

In other words, an AND expression is true only if all its inputs individually are true.

Although AND normally takes two inputs, it can take more if the whole expression is enclosed in parentheses:

```
IF (AND :AGE > 18 :AGE < 29 :SPECIES = "HORSE) [PR [WHOA!]]
```

The rule remains the same: the whole expression is true only if every input to AND is true; otherwise the expression is false:

```
?PR (AND "TRUE "TRUE "FALSE "TRUE "TRUE)
FALSE
?
```

Sometimes we may have two conditions we test for, but only one need be passed. For instance, eligibility for the Non-Average Club might consist of being taller than 76 inches or shorter than 54 inches. For this situation, use OR:

```
IF OR :HEIGHT > 76 :HEIGHT < 54 [PR [WELCOME TO THE CLUB!]]
```

Like AND, OR normally takes two inputs. If either one (or both) of the inputs is true, then the whole expression is true:

```
?PR OR "TRUE "TRUE
TRUE
?PR OR "TRUE "FALSE
TRUE
?PR OR "FALSE "FALSE
FALSE
?
```

OR, too, can take more than two inputs if the whole expression is enclosed in parentheses. Then the expression is true as long as at least one of the inputs is true:

```
?PR (OR "FALSE "FALSE "TRUE "FALSE "FALSE)
TRUE
?PR (OR 4=6 5=6 6=6 7=6 8=6)
TRUE
```

Let's use OR to patch up our SPIRAL procedure of Chapter 5. One problem it had is that eventually the pattern runs off the screen. We can change the procedure so that it stops automatically before running out of

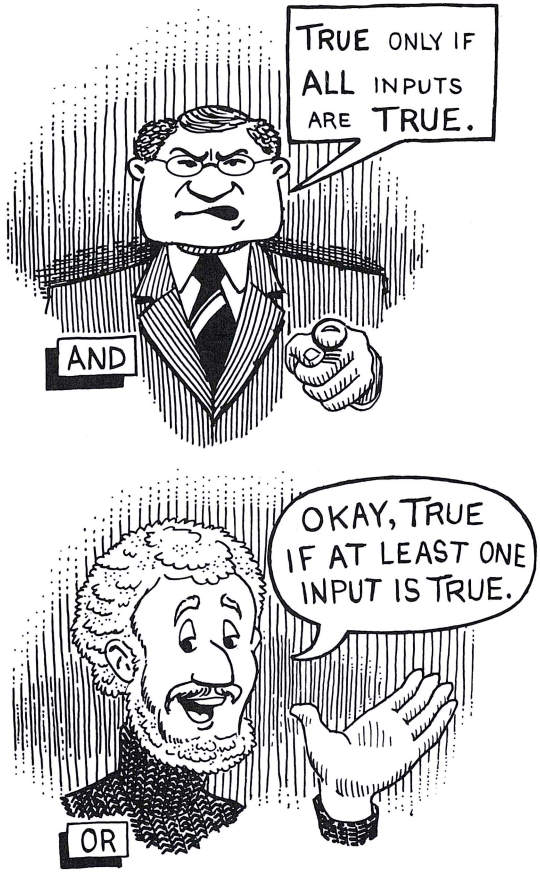


Fig. 11-2. AND vs. OR.

screen. The key is to use the Logo primitives XCOR and YCOR to find the x- and y-coordinates of the turtle. If these exceed the boundaries, then stop the procedure. The screen is 280 × 240 units, so x should be between −140 and +140, and y should be between −120 and +120. Here is a version of SPIRAL that keeps the turtle within those limits:

Listing

```
TO SPIRAL :DISTANCE :ANGLE
IF (OR XCOR < -140 XCOR > 140 YCOR < -120 YCOR > 120)
[STOP]
FD :DISTANCE
RT: ANGLE
```

```
SPIRAL :DISTANCE + 1 :ANGLE
END
```

Note that the OR combines four separate tests, so we had to enclose the OR expression in parentheses.

NOT, as we saw earlier, takes one input and negates it: "TRUE becomes "FALSE and vice versa.

Sometimes NOT can be used to simplify a more complex expression. Suppose, for instance, you want to specify that a weight be equal to or greater than 121.3 pounds. You could say

```
IF OR :WEIGHT = 121.3 :WEIGHT > 121.3 [PR [IT'S LEGAL]]
```

But being equal to or greater than 121.3 is the same as not being less than 121.3, so you could also say

```
IF NOT :WEIGHT < 121.3 [PR [IT'S LEGAL]]
```

You can build elaborate expressions using AND, OR, and NOT. Since the inputs of each are things with the value "TRUE or "FALSE, the inputs themselves can be expressions using these primitives. Suppose you are looking for a model with blond hair and blue eyes or else with black hair and green eyes. You could set up a test this way:

```
IF OR AND (:HAIR = "BLOND :EYES = "BLUE) AND (:HAIR = "BLACK
 :EYES = "GREEN) [PR [YOU ARE A CANDIDATE]]
```

There are two things to note here. One is that the command got too long to fit on one line. We will handle such instances by identing the continuation. Thus, the above should be typed as *one line,* with no RETURN until the very end. On the 40-column screen, this line will appear as

```
IF OR AND :HAIR = "BLOND :EYES = "BLUE!
AND :HAIR = "BLACK :EYES = "GREEN [PR !
[YOU ARE A CANDIDATE]]
```

with the ! marking the line continuations.

The second, more important point, is, how does Logo sort out all those predicates? It does so by counting inputs. OR expects two inputs, and the first possible input it sees is AND. But AND takes two inputs, which are the next two predicates: :HAIR = "BLOND and :EYES = "BLUE. So each of

those is evaluated as true or false. That gives the first AND expression a value of true or false, and that value becomes the first input to OR. Then Logo looks for the next input to OR and finds the second AND expression. Of course, you can always make the organization clearer by enclosing the subexpressions in parentheses:

```
IF OR (AND :HAIR = "BLOND :EYES = "BLUE) (AND :HAIR =
 "BLACK :EYES = "GREEN) [PR [YOU ARE A CANDIDATE]]
```

You *must* use parentheses if any of the ANDs or ORs have more than two inputs.

Suppose you want a model who is not a blue-eyed blonde and who is not green-eyed, black-haired. That group of candidates consists of everyone who fails the test above, so we can use the same test, but with a NOT:

```
IF NOT OR AND :HAIR = "BLOND :EYES = "BLUE AND :HAIR =
 "BLACK :EYES = "GREEN [PR [YOU ARE A CANDIDATE]]
```

As you can see, you can build quite elaborate logical constructions using AND, OR, and NOT. Here is a formal definition of AND.

AND (takes two inputs by default)
(AND . . .) (takes any number of inputs when enclosed in parentheses)

Purpose	Combines logical expressions (i.e., expressions having the value "TRUE or "FALSE), yielding a "TRUE value if all the inputs are "TRUE, and a "FALSE value otherwise
Abbrev	None
Inputs	Any Logo data object or output having the value "TRUE or "FALSE
Output	The value "TRUE if all inputs are "TRUE; the value "FALSE otherwise

Examples:

```
?PR AND 3 > 7 4 > 2
FALSE
?PR (AND 4 = 4 9 > 2 1 < 7)
TRUE
?
```

We'll summarize OR and NOT in a table later. Now let's look at the variant of IF.

11.4 The Primitives TEST, IFTRUE, And IFFALSE

Sometimes an IF along with all its inputs can take up quite a bit of space and become hard to read. Logo offers an alternative to IF that is a bit more flexible and spreads the information over more lines. This approach uses the primitives TEST, IFTRUE, and IFFALSE. Roughly, these correspond to the first, second, and third inputs respectively of an IF statement. For example, consider this IF statement:

```
IF :NAME = "HILARY [GREET.HILARY] [REJECT]
```

(Here GREET.HILARY and REJECT are some appropriate procedures). We can rewrite this as

```
TEST :NAME = "HILARY
IFTRUE [GREET.HILARY]
IFFALSE [REJECT]
```

TEST does what its name suggests; it tests its input to see if it is true or false. Logo then remembers the result. When an IFTRUE (abbreviation: IFT) shows up, its input is executed if the TEST was true. Similarly, the IFFALSE (abbreviation: IFF) has its input executed if the preceding TEST was false.

TEST takes one input. Like the first input to IF, this input must be a Logo data object or output having the value "TRUE or "FALSE.

IFTRUE and IFFALSE each take one input. This input should be an action or list of actions enclosed in braces. You can have several IFTRUEs and IFFALSEs in a procedure. They will always refer back to the latest TEST. Consider this fragment:

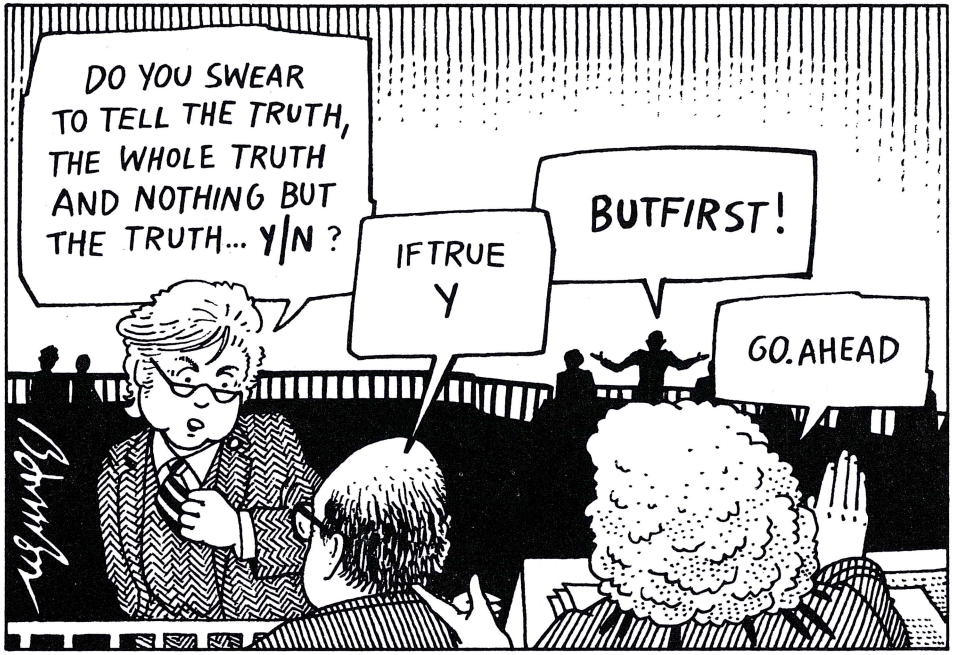
```
TEST :SCORE > 89
(PR [YOUR SCORE WAS] :SCORE)
IFT [PR [THAT'S AN A]]
IFF [PR [I THINK YOU CAN DO BETTER THAN THAT]]
MAKE "AVG FIND.AVERAGE
IFT [(PR [YOUR SCORE WAS] :SCORE - :AVG [HIGHER THAN
 AVERAGE])]
```

```
TEST :NAME = [IAGO O'BLARNEY]
IFT [(PR :NAME [PLEASE REPORT TO THE DEAN]]
```

The IFF and the first two IFT's all refer to the first TEST, i.e., is the score greater than 89? The final IFT refers to the second TEST, since that is the last TEST made before that IFT is reached. Note that other statements can be interspersed between IFTs, IFFs, and a TEST.

The TEST-IFTRUE-IFFALSE approach is particularly useful if the test or the actions are lengthy.

Now let's take a closer look at Logo predicates, that is, at those Logo primitives that output true-false values.



11.5 Predicates

Logo predicate primitives share one feature: they all output either "TRUE or "FALSE. Most follow the usual pattern for procedures, but the three we have discussed (=, >, and <) are a little unusual. They take two inputs, but unlike other procedures, these take the first input before the procedure name and the second input afterwards. For example, consider

```
?PR 5 = 6
FALSE
?
```

Here 5 is the first input to =, 6 is the second, and the output is "FALSE.

Logo offers another version of = that is closer to Logo traditions. It is EQUALP. The chief difference between EQUALP and = is that EQUALP takes its input in the usual place: after the procedure name. Thus

```
?PR EQUALP 5 6
FALSE
?
```

works just like the preceding example. In both cases the inputs are 5 and 6, and they are tested to see if they are the same. They are not, so "FALSE is outputted.

The "EQUAL" part of the name is easy enough to understand, but what about the "P"? The "P" at the end is to remind us that EQUALP is a predicate procedure. Logo has a whole family of such procedures, and we will look at some of them now.

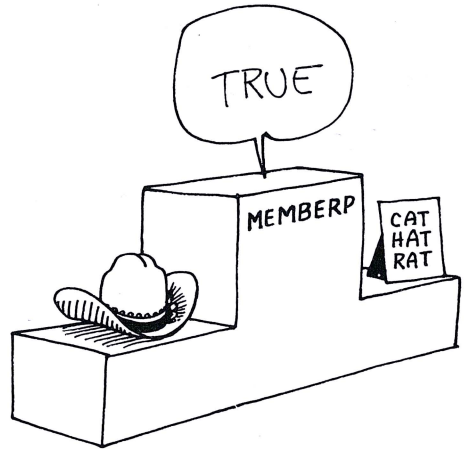
One of the most useful predicates is MEMBERP. It takes two inputs. The first is an item, the second a list; MEMBERP checks if the item is a member of the list.

```
?PR MEMBERP "BOB [TED JACK BOB DICK]
TRUE
?PR MEMBERP "B [TED JACK BOB DICK]
FALSE
?
```

Note that is not enough that the list has a B somewhere in it. The first input item has to match a *complete* item from the list. The items must match in type, too:

```
?PR MEMBERP [BOB] [TED JACK BOB DICK]
FALSE
?PR MEMBERP [BOB] [[TED] [JACK] [BOB] [DICK]]
TRUE
?
```

The *list* [BOB] does not match the *word* "BOB, but it does match the *list* [BOB].



MEMBERP "HAT [CAT HAT RAT]

Fig. 11-3. MEMBERP on the job.

Logo has many predicates. Table 11-1 summarizes the predicates we have discussed plus several more.

Table 11-1. Predicate Primitives (All Output "TRUE or "FALSE)

A. COMPARISONS

=	Takes two inputs, one before and one after the = symbol. Yields "TRUE if both inputs are the same. ?PR 5 = 2 +3 TRUE ?PR [DOG FOOD] = FIRST [[DOG FOOD] [DOD FOOG]] TRUE
EQUALP	Same as =, but both inputs follow. ?PR EQUALP "HANK "HARK FALSE
>	Takes two inputs, one before and one after the > symbol. Each input must be a number or yield a numerical value. Yields "TRUE if the first input is greater than the second. ?PR 8 > 2 * (3 + 4) FALSE
<	Same as >, but yields "TRUE if the first input is less than the second.

```
?PR 8 < 2 * (3 + 4)
TRUE
```

B. MEMBERSHIP PREDICATES

MEMBERP Takes two inputs. The first is an item, the second a list. Yields "TRUE if the item is in the list.

```
?PR MEMBERP ROSE [SHE ROSE AT DAWN]
TRUE
```

WORDP Takes one input. Yields "TRUE if the input is a word.

```
?PR WORDP FIRST [THE GATORS EYES GLAZED]
TRUE
```

LISTP Takes one input. Yields "TRUE if the input is a list.

```
?PR LISTP "OSMICS
FALSE
```

NUMBERP Takes one input. Yields "TRUE if it is a number.

```
?MAKE "FUMET 113
PR NUMBERP :FUMET
TRUE
PR NUMBERP "FUMET
FALSE
```

EMPTYP Takes one input. Yields "TRUE if the input is [ ] (the empty list) or " (the empty word)

C. INPUT PREDICATES

KEYP Takes no inputs. Outputs TRUE if keyboard character is pending

BUTTONP For paddle users. Takes 0, 1, or 2 as an input. Outputs "TRUE if the button on the corresponding paddle is pressed.

D. NAME PREDICATES

DEFINEP Takes one input. Outputs "TRUE if the input is the name of a procedure:

```
?PR DEFINEP "FORWARD
TRUE
```

NAMEP Takes one input. Outputs "TRUE if the input is the name of something having a value:

```
?MAKE "BLOP 23
PR NAMEP "BLOP
TRUE
```

E. LOGICAL PREDICATES

AND Takes 2 (by default) or more (enclose AND and all inputs in paren-

theses) inputs. Inputs must all have either "TRUE or "FALSE as values. Output is "TRUE if and only if all inputs are "TRUE.
?PR AND 3 > 1 + 6 1000 > 27
FALSE

OR — Takes 2 (by default) or more (enclose AND and all inputs in parentheses) inputs. Inputs must all have either "TRUE or "FALSE as values. Output is "TRUE if and only if at least one input is "TRUE.
?PR AND 3 > 1 + 6 1000 > 27
TRUE

NOT — Take one input, which must have a value of "TRUE or "FALSE. Output is "TRUE is the input is "FALSE, and vice versa.
?PR NOT 18 = 10 + 8
FALSE

Let's see how we can use some of these predicates.

11.6 The Utility GETB.NUMBER: Fetching a Number

Many interactive programs at some point ask the user to type in a number. We can use READLIST to get the user's response, but READLIST is a little bit indiscriminate. The user might type SIX instead of 6, and READLIST would accept it. But when, say, the program tried to multiply SIX by 3, Logo will protest. Multiplication wants numbers not words. It would be useful, then, to develop a utility that will pass numbers only on to a program. GETB.NUMBER is one answer to this need. (The "B" in GETB.NUMBER is a code we are using to indicate that the procedure checks for errors. See the next chapter for more on this topic.) Here's the listing:

Listing

```
TO GETB.NUMBER
LOCAL "ANS
MAKE "ANS READLIST
IF :ANS = [ ] [PR [A NUMBER PLEASE, THEN <RETURN>] OP
 GETB.NUMBER]
IF NOT NUMBERP FIRST :ANS [PR [THAT IS NOT A NUMBER, TRY
 AGAIN] OP GETB.NUMBER
PR [ ]
```

```
OP FIRST :ANS
END
```

There are two IFs. The first test to see if the response was the empty list. This would occur if the user hit RETURN and nothing else. If this happens, the user is prompted to do it right, and GETB.NUMBER is repeated. (We will discuss the mechanism in a moment.)

The second test checks to see if the response is not a number and has the user try again if it isn't. Notice that we use FIRST :ANS and not :ANS. The reason for this is that READLIST returns a LIST. If the user types in 41, then :ANS has the value [41], and a list like that is never a number. FIRST :ANS, however, would be the first element of the list, that is, the number 41. So it is FIRST :ANS that we must check with MEMPERP. Incidentally, FIRST does not like the empty list, which is why we had to check for that first.

If the user's response clears all the hurdles, the program outputs the number. Thus other procedures can use GETB.NUMBER in this fashion:

```
MAKE "ELEPHANTS GETB.NUMBER
```

The procedure guarantees that its output will be a number.

But what about if something goes wrong? Why do we say

```
[OP GETB.NUMBER]
```

instead of simply

```
[GETB.NUMBER]
```

in the IF statements? Remember how OUTPUT works. When used in a procedure, it outputs a value to the procedure that called it. If we have a recursive series of procedures, then each one has to output its value back to the procedure that called it until the original procedure is reached. Let's go back to

```
MAKE "ELEPHANTS GETB.NUMBER
```

to see how this works. If we type a 3, then GETB.NUMBER outputs a 3, and :ELEPHANTS is equated to that value immediately. Suppose, however, we type THREE first. Then :ELEPHANTS is equated not to THREE BUT TO GETB.NUMBER, since that is what is outputted. If we then type

3, the second running of GETB.NUMBER outputs a 3 which then is assigned to :ELEPHANTS. In effect, the lower level of GETB.NUMBER transmits the 3 to the higher level, which transmits it to the original calling procedure.

Earlier we presented a program GUESS.ME in which the user tries to guess the secret number, which was 4. Let's create a slightly more elaborate version in which the secret number is chosen at random and in which the player is told whether her guess is high or low. We can use GETB.NUMBER to fetch the guess. We call this version GUESS.NUMBER:

Listings

```
TO GUESS.NUMBER
LOCAL "ANS
MAKE "ANS (RANDOM 100) + 1
PR [GUESS MY SECRET NUMBER!]
PR [IT IS IN THE RANGE 1-100]
GETGUESSES :ANS
END


TO GETGUESSES :ANS
LOCAL "GUESS
MAKE "GUESS GETB.NUMBER
IF :GUESS > :ANS [PR [TOO HIGH! TRY AGAIN]
IF :GUESS < :ANS [PR [TOO LOW! TRY AGAIN]
IF :GUESS = :ANS [PR [THAT'S IT! YOU WIN!] STOP]
GETGUESSES :ANS
END
```

Here is a sample run:

Sample run

```
?GUESS.NUMBER
GUESS MY SECRET NUMBER!
IT IS IN THE RANGE 1-100
50
TOO LOW! TRY AGAIN
75
TOO HIGH! TRY AGAIN
SIXTY
```

```
THAT IS NOT A NUMBER. TRY AGAIN
60
THAT'S IT! YOU WIN!
?
```

The structure we use for GETB.NUMBER is a common one. For example, suppose we want to fetch not a number, but a word. Then we just have to change GETB.NUMBER a little:

Listing

```
TO GETB.WORD
LOCAL "ANS
MAKE "ANS READLIST
IF :ANS = [ ] [PR [A WORD PLEASE, THEN <RETURN>] OP
 GETB.WORD]
OP FIRST :ANS
END
```

Since READLIST produces a simple list, FIRST :ANS is a word automatically. We just have to screen out the empty list return. We'll say more about these utilities in the next chapter.

Now let's take a more graphic angle.

11.7 STEER: Fly Your Turtle with GET.KEY

Here we will use the KEYP predicate to let us guide the turtle interactively. First, consider this program:

```
TO MOVE
FD 2
MOVE
END
```

The turtle moves two units, the procedure calls itself by recursion, the turtle moves another two units, and so on until you hit CONTROL-G to stop the action.

Well, this is a rather boring program, but it gets more interesting if we can tell the turtle to change its direction. To do that, we need to have the program check to see if there are any instructions coming in from the keyboard while the turtle is moving. Here's one way to do it:

Listing

```
TO STEER
MAKE "DIR GET.KEY
IF :DIR = "S [LT 30]
IF :DIR = "D [RT 30]
FD 2
STEER
END


TO GET.KEY
IF KEYP [OP RC] [OP " ]
END
```

First, note how GET.KEY works. Using KEYP, a predicate, it checks to see if anything has been typed in at the keyboard. If a key has been struck, the procedure outputs that character; otherwise it outputs the empty word. Now look at STEER. Each cycle, before the turtle moves its two units, STEER uses GET.KEY to see if there has been keyboard input. If an "S was typed, the turtle veers left 30 degrees, and if a "D was typed the turtle veers right. Otherwise the turtle maintains its heading. (We could have used "L and "R instead of "S and "D, but we thought is might be confusing, since the L-key is on the right side of the keyboard and the R-key is on the left side!)

Try it. If the turtle moves too slowly for your tastes, increase the distance it moves each cycle.

11.8 Make Your Own Predicates

Logo predicates, we have seen, output a value of "TRUE or "FALSE. We can create our own predicates by making procedures that output only these values. For example, in our revised version of SPIRAL, we tested to see if the turtle is out of bounds:

Listing

```
TO SPIRAL :DISTANCE :ANGLE
IF (OR XCOR < -140 XCOR > 140 YCOR < -120 YCOR > 120)
   [STOP]
```

```
FD :DISTANCE
RT: ANGLE
SPIRAL :DISTANCE + 1 :ANGLE
END
```

We can rewrite it this way:

Listing

```
TO SPIRAL :DISTANCE :ANGLE
IF OUTOFBOUNDS? [STOP]
FD :DISTANCE
RT: ANGLE
SPIRAL :DISTANCE + 1 :ANGLE
END


TO OUTOFBOUNDS?
IF XCOR < -140 [OUTPUT "TRUE]
IF XCOR > 140 [OUTPUT "TRUE]
IF YCOR < -120 [OUTPUT "TRUE]
IF YCOR > 120 [OUTPUT "TRUE]
OUTPUT "FALSE
END
```

Here we followed a common convention of ending the name of a user-defined predicate with a question mark. This helps remind us that it is a true-false procedure. We could have used a P (for predicate) instead for ending the program name, but by using a question mark we distinguish between primitive predicates and user-defined predicates.

This approach yields a program that is a little easier to follow. The name of the procedure (OUTOFBOUNDS?) makes it pretty clear that the IF statement is all about. (However, calling up a second procedure does slow the program down a bit.)

11.9 Summary

Here are the important points in this chapter.

1. The Logo conditionals (IF, TEST, IFTRUE, IFFALSE) and the

Logo predicates give us the means to construct program that make decisions.

2. Logo predicates are expressions, primitives, or procedures that have as the only possible values the words "TRUE and "FALSE.
3. The Logo logical operators (AND, OR, NOT) take predicates as inputs and produce a "TRUE or "FALSE output.
4. We can use predicates to create utilities like GETB.NUMBER.
5. We can make our own predicate, that is, procedures that output a "TRUE or a "FALSE.

11.10 Review Questions and Exercises

Review Questions

1. In the procedure QUERY, what happens if the user responds with something other than Y or N?
2. Indicate for each case whether TRUE or FALSE is printed:
   a. PR MEMBERP 2 + 5 [3 7 9 10]
   b. MAKE "FLOODGE "MICKEY
      PR NOT LISTP :FLOODGE
   c. MAKE "LEGS 20
      PR OR :LEGS > 40 :LEGS / 2 < 12

Exercises

1. Formulate formal definitions for OR and NOT.
2. Write your name using STEER2
3. Write a predicate that checks to see if its input is a vowel. (Use MEMBERP.)
4. Improve GUESS.NUMBER by having the program keep score.
5. Devise a RACETRACK program using STEER. Have the program draw two concentric circles for the track and place the turtle on the track. Have the program tell the user to type Y when she is ready to start. Once the Y is typed, start the turtle up and let the user steer it around the track.

Answers to Review Questions

(1) Any answer beginning with Y (YES, YUP, YOYO, etc.) works the same as a simple Y. A RETURN without any other response crashes the program, for FIRST does not like any empty list. Any other response has the same effect as a simple N. (2a) TRUE (2b) TRUE (2c) TRUE

CHAPTER 12

HUMANIZING YOUR PROGRAMS

CONCEPTS:

interactive program writing
error trapping
screen formatting
adding remarks

12.0 Introduction

In the first six chapters we built a framework for learning Logo by emphasizing primitives and their inputs. Then we added to our structure by learning about Logo's data objects, the use of variables and program control. We have written simple programs to control the turtle and to manipulate numbers, words and lists.

In this chapter, we begin Part 3 of our book: How to plan and write programs. We'll begin by presenting a hypothetical dialogue between a program and a new user. This example illustrates some things *not* to do with a program. See if you can spot the problems.

Sample Program #1

```
ENTER NAME
BRANDON FRUPP
NOW ENTER LAST NAME
FRUPP
HELLO BRANDON FRUPP FRUPP
```

```
ENTER BIRTHDATE
11/16/52
NOT THAT WAY MUSHBRAIN, TRY AGAIN
NOVEMBER 16, 1952
NOT THAT WAY MUSHBRAIN, TRY AGAIN
11-16-52
TOOK YOU LONG ENOUGH, DONKEY BREATH
ENTER PHONE NUMBER: AND GET IT RIGHT
OR YOU'RE IN BIG TROUBLE.
457-2467
ERROR! ERROR! ERROR! . . . PROGRAM DUMP!#"!?#$??%0*=!?


AWK?
```

If you feel that this example epitomizes good program style, then read no further, for our intent in this chapter is to free programs from these features. The kind of programs that we want to write work like this:

Sample Program #2

```
HELLO.
THIS PROGRAM CREATES A STORY WITH YOU
AS ITS MAIN CHARACTER.
WHAT IS YOUR NAME?
SUZY
THAT'S A NICE NAME. OKAY, SUZY, WHAT IS YOUR AGE?
SIXTEEN
I'M SORRY. I'M NOT ALL SMART AND I DON'T UNDERSTAND THAT.
CAN YOU ENTER YOUR AGE AS A NUMBER, LIKE 12 OR 56.
16
OKAY SUZY, YOU ARE 16. DID I GET THAT RIGHT? (YES OR NO?)
YES
THANK YOU SUZY, YOU SURE ARE EASY TO TALK TO.
```

This is an example of a *humanized* program.

The goal of such a program is to make the user feel as if she is interacting with a friendly, helpful person who has her best interests at heart. With a humanized program, the user isn't forced to remember a complex series of steps that must be executed in an exact manner. Instead, the program guides the user through the proper steps. If the user responds incorrectly, a humanized program doesn't crash disastrously; instead, it gives the user

another chance, perhaps even pointing out what errors to avoid. A humanized program produces a sense of give-and-take that makes using it a pleasant activity, not a technological ordeal.

Humanized programming reflects a concern for the user, but let's not forget the programmer. The same friendly, helpful attitude can make the programmer's life easier, too. Good systematic work habits make it simpler for a programmer to develop and maintain programs.

Thus humanized programming makes life nicer for users and for programmers. This sounds like a worthwhile goal; how do we implement it? Here are six important elements of humanized programming, three to aid the user, and three to aid the programmer:

User Aids

1. Interactive programming
2. Error trapping
3. Formatting

Programmer Aids

1. Top-down program design
2. Using tools or utilities
3. Good documentation

We will see examples of all six points in this chapter. Let's begin with a look at interactive programming.

12.1 What's an Interactive Program?

An interactive program creates a dialogue between the computer and the user (Recall SPI.MENU and GUESS.NUMBER). As the program runs, it can communicate results and requests for information to the user, and the user can provide the program with direction and information. Suppose, for instance, you have prepared a program that converts a weight in pounds to kilograms. You could make a relatively noninteractive procedure like this:

Listing

```
TO KILO :POUNDS
PRINT :POUNDS / 2.2
END
```

A sample run could look like this:

```
?KILO 220
100
```

This works, but it is not very friendly. The user has to remember what KILO does and that it takes one input, the number of pounds to be converted. What we want is a program (call it KILOS) that works like this:

Sample Run

```
?KILOS
HI! I CAN CONVERT WEIGHT IN POUNDS TO KILOGRAMS.
PLEASE TYPE IN THE NUMBER OF POUNDS.
220
THAT'S 100 KILOGRAMS.
```

This version tells us what the program does, prompts the user to type in the desired information, and explains the answer it prints out. Ah, much better. These are the features we want in an interactive program—but how do we do it? It's easy; we use PRINT to let the computer talk to the computer. Here is a primitive version of KILOS:

Listing

```
TO KILOS
PRINT [HI! I CAN CONVERT WEIGHT IN POUNDS TO KILOGRAMS.]
PRINT [PLEASE TYPE IN THE NUMBER OF POUNDS.]
MAKE "KILO (FIRST READLIST) / 2.2
PRINT (SENTENCE [THAT'S] :KILO [:KILOGRAMS.])
END
```

For our sample run, READLIST produced the list [220], and FIRST READLIST extracts the number 220 from the list. Then MAKE assigned the value 100 (i.e., 220 / 2.2) to the variable "KILO.

Why is this version primitive (aside from the fact that it uses only Logo primitives)? It is primitive because it messes up if the user types in the wrong kind of information:

Sample run

```
?KILOS
HI! I CAN CONVERT WEIGHT IN POUNDS TO KILOGRAMS.
```

```
PLEASE TYPE IN THE NUMBER OF POUNDS.
FIFTY
/ DOESN'T LIKE "FIFTY AS INPUT.
```

This unfriendly behavior can be fixed by using error trapping, and that will be our next topic.

Meanwhile, there is one more interactive feature we would like to add. Let's modify the program so that it repeats, or "echoes," the user's input. This reassures the user that the computer is processing the right information. Here's one way to do that:

Listing

```
TO KILOS
PRINT [HI! I CAN CONVERT WEIGHT IN POUNDS TO KILOGRAMS.]
PRINT [PLEASE TYPE IN THE NUMBER OF POUNDS.]
MAKE "POUNDS FIRST READLIST
PRINT (SENTENCE [THAT'S] :POUNDS [POUNDS.])
PRINT (SENTENCE :POUNDS [ARE] :POUNDS/2.2 [KILOGRAMS.])
END
```

We created a new variable :POUNDS to store the user's input so that we could later print the value out.

Sample Run

```
?KILOS
HI! I CAN CONVERT WEIGHT IN POUNDS TO KILOGRAMS.
PLEASE TYPE IN THE NUMBER OF POUNDS.
220
THAT'S 220 POUNDS.
220 POUNDS ARE 100 KILOGRAMS.
```

In summary, an interactive program lets the user and a running program exchange information. Useful interactive features include the following:

1. An explanation of what the program does.
2. Instructions to the user about what he should do next.
3. Input checking to see if the user's input is in the right form.
4. An "echoing" of the user's input, showing that the program received the right information.

5. An identification or explanation of whatever answers the program prints out.

Now, let's turn now to the third point on this list.

12.2 Error Trapping

The process of checking input is called "error trapping." Error trapping is the art of anticipating what errors a user might make and then heading off these errors before they can do any harm.

There are two main approaches. The first approach (positive error checking) is to decide in advance what the possible correct user responses should be. The program checks to see if the actual response belongs to the group of correct responses. If it does, fine. Otherwise, the program goes back to give the user another try.

The second approach (negative error checking) is to decide in advance what the possible wrong responses are. If the actual response belongs to this group, the program gives the user another try. Otherwise, the program proceeds normally.

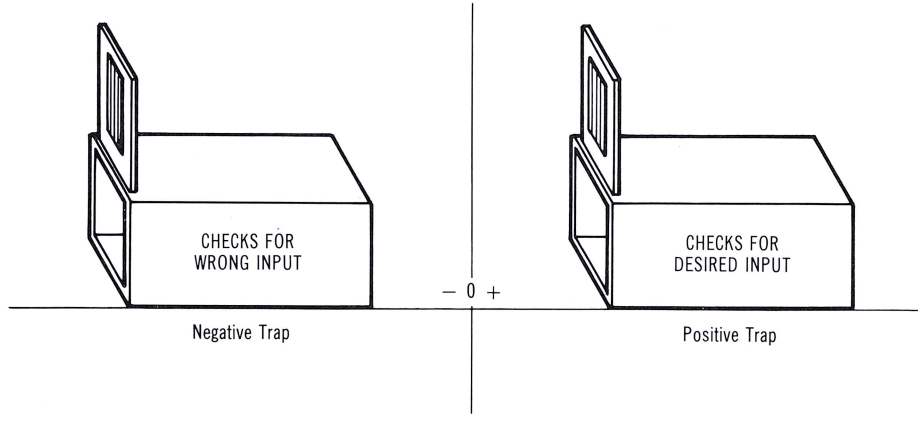


Fig. 12-1. Positive and negative error trapping.

You can use either or both approaches, depending on what seems most reasonable for the situation. For instance, in our KILOS example, we can use the GETB.NUMBER procedure we developed in the preceding chapter. It uses negative error checking to screen out the case of the user hitting the RETURN key without entering any data, and it uses positive error

checking to see if the input is indeed a number. To refresh your memory, here is the listing for GETB.NUMBER.

```
TO GETB.NUMBER
LOCAL "ANS
MAKE "ANS READLIST
IF :ANS = [ ] [PRINT [A NUMBER PLEASE, THEN <RETURN> ]
   OUTPUT GETB.NUMBER]
IF NOT NUMBERP FIRST :ANS [PRINT [THAT IS NOT A NUMBER, TRY
   AGAIN] OUTPUT GETB.NUMBER]
PRINT [ ]
OUTPUT FIRST:ANS
END
```

Hmm, this is just what we need to include error checking in our KILOS program. Here is a revised version:

```
TO KILOS
PRINT [HI! I CAN CONVERT WEIGHT IN POUNDS TO KILOGRAMS.]
PRINT [PLEASE TYPE IN THE NUMBER OF POUNDS.]
MAKE "POUNDS GETB.NUMBER
PRINT (SENTENCE [THAT'S] :POUNDS [POUNDS.])
PRINT (SENTENCE :POUNDS [ARE] :POUNDS/2.2 [KILOGRAMS.])
END
```

All we had to do was to replace FIRST READLIST with the more advanced GETB.NUMBER. Now a sample run could look like this:

Sample run

```
?KILOS
HI! I CAN CONVERT WEIGHT IN POUNDS TO KILOGRAMS.
PLEASE TYPE IN THE NUMBER OF POUNDS
TWO HUNDRED TWENTY
THAT IS NOT A NUMBER, TRY AGAIN
220
THAT'S 220 POUNDS.
220 POUNDS ARE 100 KILOGRAMS
```

The procedure GETB.NUMBER is an example of a tool or utility, a handy procedure that meets a frequent need in programming. Many programs need to input numbers, and we can use the GETB.NUMBER tool for

the task. There is no need to design a new number-reading procedure every time we construct a new program.

Let's look at two more utilities designed to meet the needs of interactive programming.

12.3 Two New Input Utilities: GETB.YN and GETC.NUMBER

Often, interactive programs request the user to respond with a simple Y or N to a yes-no question. Our first new utility, GETB.YN, handles that response.

Just what do we want GETB.YN to do? First, we want to make sure the user has typed an answer and not just the RETURN key. (An empty list would foul up the rest of the procedure.) Secondly, we want to check if the response is either a Y or a N. If either of this first two tests is not met, give the user another chance. Finally, once the user is successful, the procedure should output the user's response. Here is one version:

```
TO GETB.YN
LOCAL "ANS
MAKE "ANS READLIST
IF :ANS = [ ] [PR [ENTER Y OR N, THEN <RETURN>] OP GETB.YN]
IF NOT MEMBERP FIRST :ANS [Y N] [PR [Y OR N? TRY AGAIN] OP
GETB.YN]
OP FIRST :ANS
END
```

The second line makes the variable "ANS local, and the third line makes "ANS equal to whatever response the user has typed. The fourth line tells the computer what to do if the user has responded by hitting just RETURN: tell the user what to do, and use recursion (by outputting the procedure name) to read in a new response.

The second IF statement uses the MEMBERP primitive to see if the user's response belongs to a list of acceptable responses. In this case the list was [Y N], but the list could be extended to include YES and NO, too. Notice that we compare FIRST :ANS and not :ANS to the list. Remember, :ANS itself is a list (perhaps [Y], for example), so we use FIRST, to extract the first word from the list (Y, for example). Then this word can be compared to the words in the check list. Again, recursion is used to give the user another chance.

Finally, when the user types in an acceptable response, the line OP FIRST :ANS sends the response on to the procedure calling GETB.YN.

Here is an example using our new tool.

```
?COMP
DO YOU LIKE COMPUTERS? (Y OR N)
I DON'T KNOW WHAT TO SAY
Y OR N? TRY AGAIN
Y
THANK YOU, I NEEDED THAT!
```

Here is the procedure that demonstrates the utility:

```
TO COMP
PR [DO YOU LIKE COMPUTERS? (Y OR N)]
LOCAL :ANS
MAKE "ANS GETB.YN
IF :ANS = "Y [PR [THANK YOU, I NEEDED THAT!]]
IF :ANS = "N [PR [DARN!]]
END
```

Okay, that seems to work. Let's take a look at the next utility, GETC.NUMBER. You probably can guess from its name that it fetches a number. We already have GETB.NUMBER to do that, so why a new procedure? The answer is that we will add a new level of error checking. GETC.NUMBER also lets the user check his input and change it if he wants. Here, for instance, is a sample run of a procedure that uses GETC.NUMBER:

```
?FLOPPY
HOW MANY FLOPPY DISKS DO YOU WANT?
SOME
THAT IS NOT A NUMBER, TRY AGAIN
100


IS THIS RIGHT? Y OR N?
N


ENTER NEW NUMBER
10


IS THIS RIGHT? Y OR N?
```

```
Y
OKAY, 10 FLOPPIES COMING UP!
```

Here is the procedure that tests out the new utility. Procedures that are used to "test drive" utilities or programming concepts are often called "drivers." Here is the driver for GETC.NUMBER, a procedure called FLOPPY.

```
TO FLOPPY
PR [HOW MANY FLOPPY DISKS DO YOU WANT?
MAKE "DISKS GETC.NUMBER
(PR [OKAY,] :DISKS [FLOPPIES COMING UP!])
END
```

And here is our new utility:

```
TO GETC.NUMBER
LOCAL "ANS
MAKE "ANS READLIST
IF :ANS = [ ] [PR [PLEASE ENTER A NUMBER, THEN <RETURN>] OP
   GETC.NUMBER]
IF NOT NUMBERP FIRST :ANS [PR [THAT IS NOT A NUMBER, TRY
   AGAIN] OP GETC.NUMBER]
PR[ ]
PR [IS THIS RIGHT? Y OR N?]
IF GETB.YN = "Y [OP FIRST :ANS] [PR [ENTER NEW NUMBER] OP
   GETC.NUMBER]
END
```

Notice that GETC.NUMBER makes use of GETB.YN. Also notice that the final IF statement uses the two-alternative form.

These two useful examples provide models for other interactive procedures. Using them as guides, you can construct other procedures such as GETB.WORD, GETC.WORD, GETB.LIST, and GETC.LIST. The first two procedures, of course, would fetch a word from the user, and the second two would fetch a list. Following the naming convention we have been using, the B-level procedures would provide error checking, and the C-level procedures would, in addition, let the user review and change his input.

12.4 Sample Program: Math Quiz

What could be more fun than testing our knowledge of the multiplication tables? Well, perhaps that isn't at the top of the list, but it still makes for a

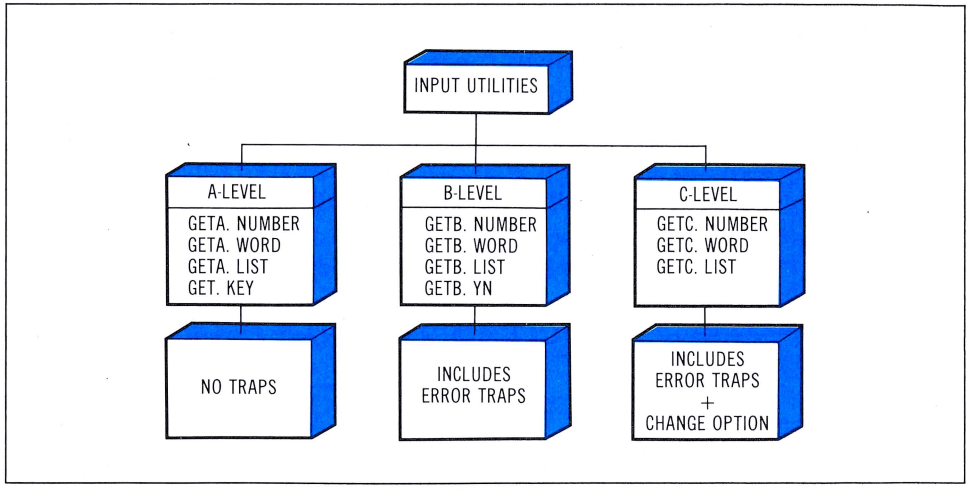


Fig. 12-2. A B and C level utilities.

good, interactive program. Before we rush off and write such a program, however, it is a good idea to think carefully about what exactly we want the program to do. Then we can proceed to the hows of doing it.

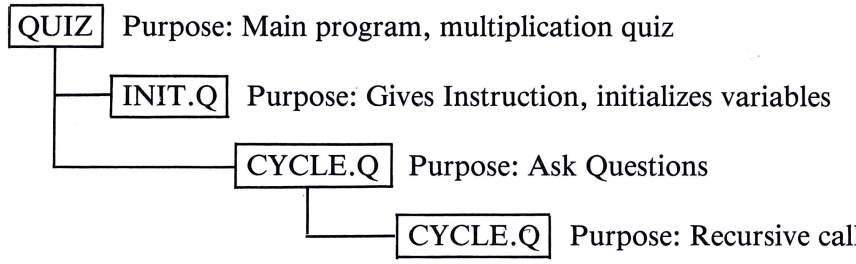
While we mull over what the program should do, our thoughts could proceed this way. The most basic program would print out two numbers, ask you what the product is, and check to see if you have the right answer.

Already, we face a decision: which two numbers? Let the computer decide; it can use the Logo RANDOM function to choose the two numbers. Another decision is how many questions will the program ask. Let's delegate that decision, too, and let the user tell the program when she wants to stop. What range of numbers should we use? Let's start easy, using numbers from 1 to 10. To let the user know how she is doing, we'd better keep track of the number of questions and the number of right answers. Oh, yes, and we'd better have the program explain its rules to the user.

By now the program has quite a list of things to do; in fact, we have more to do than we can easily keep in mind at one time. At this point it is a good idea to steer clear of the details and prepare a broad, overall plan. Once we have the big picture, we can look at the individual parts and develop them as necessary.

Okay, what is the big picture for this program? First, let's think about the overall structure. Some things need be done only once, when the program begins. For example, at the beginning of the program, we print an explanation and define variables to keep track of the number of questions and the number of right answers. Other parts of the program, such as

choosing numbers and checking the user's answer, will be repeated until the user decides to stop. This suggests breaking up the program into two parts, one which repeats and one that doesn't. Let's call our program QUIZ; here is our first pass using the box chart.



Here is the listing for QUIZ:

```
TO QUIZ
INIT.Q
CYCLE.Q
END
```

The INIT.Q procedure will start the program off. The CYCLE.Q procedure will contain the actual quiz action, and it will use recursion to repeat itself. We can write INIT.Q without further ado:

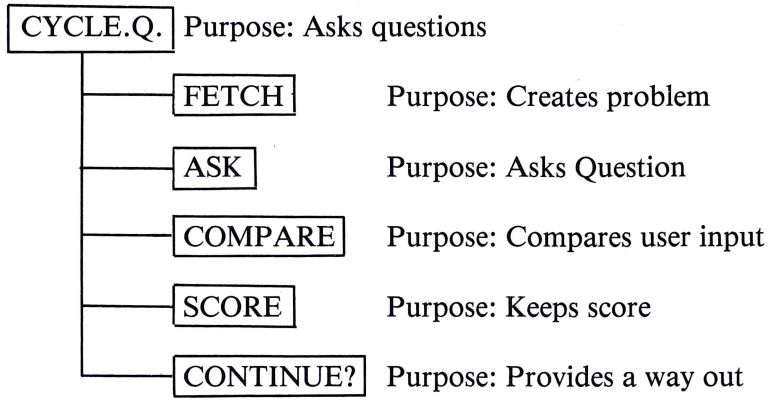
```
TO INIT.Q
PR [HOW'S YOUR MULTIPLICATION? I WILL QUIZ]
PR [YOU ON THAT. JUST ENTER AN ANSWER TO]
PR [EACH QUESTION, AND YOU'LL BE TOLD IF]
PR [YOU'RE RIGHT OR NOT. I WILL KEEP A]
PR [RUNNING SCORE]
MAKE "QUESTIONS 0
MAKE "CORRECT 0
END
```

CYCLE.Q is more involved. Before writing it, let's prepare a step-by-step list of what we want this procedure to do:

1. Fetch two numbers in the 0 to 10 range along with their product.
2. Print the two numbers and ask the user to type in his response.
3. Compare the response to the true answer, tell the user whether or not she was right, and update the totals for questions and correct answers.

4. Calculate and print the user's running score.
5. Continue with the questioning if the user wants to.

Of course, this is a bit arbitrary, for we could have combined some steps together and subdivided other steps. The important point, however, is that we have broken the task down into smaller, more manageable parts. The simplest thing to do next, then, is use a separate procedure for each of the steps in the list as shown in the box chart below:



Here is the listing for CYCLE.Q

```
TO CYCLE.Q
FETCH
ASK
COMPARE
SCORE
CONTINUE?
END
```

Now we're ready to try a first draft of each of our subprocedures:

```
TO FETCH
MAKE "NUM1 RANDOM 10
MAKE "NUM2 RANDOM 10
MAKE "ANS :NUM1 * :NUM2


TO ASK
PR (SE [WHAT IS] :NUM1 [TIMES] :NUM2 [?])
MAKE "RESPONSE GETB.NUMBER
```

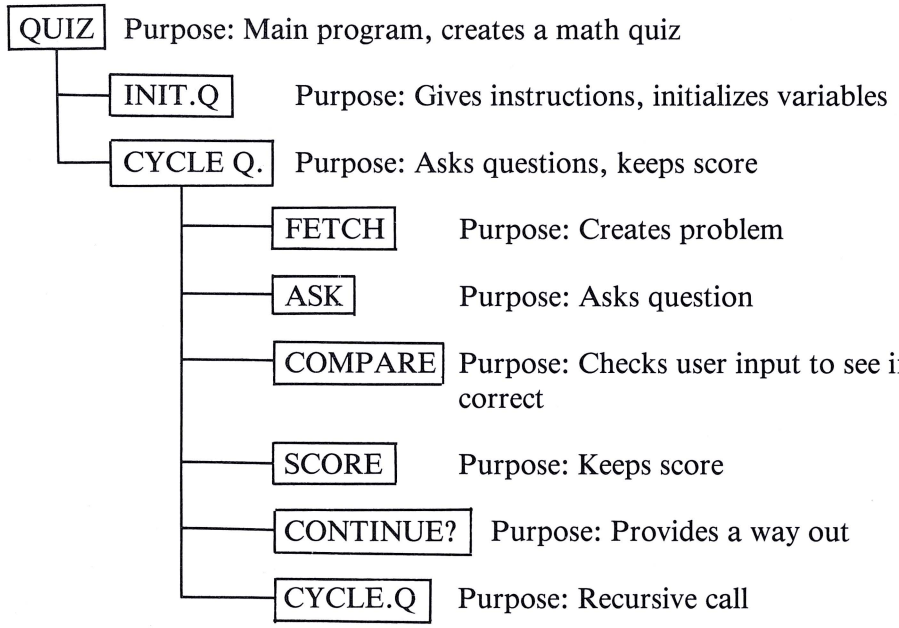
Notice we use our utility GETB.NUMBER

```
TO COMPARE
IF :ANS = :RESPONSE [PR [THAT'S RIGHT!] MAKE "CORRECT
   :CORRECT + 1] [(PR [NOPE, THE ANSWER WAS ] :ANS [.])]
MAKE "QUESTIONS :QUESTIONS + 1
END


TO SCORE
PR (SE [YOU'VE] :CORRECT [OUT OF] :QUESTIONS [RIGHT.
   THAT'S]
100 * :CORRECT / :QUESTIONS [/%.])
END
```

Here we give the percentage of correct answers.

```
TO CONTINUE?
PR [DO YOU WISH TO CONTINUE? < Y OR N >]
IF GETB.YN = "N [PR [THANKS FOR THE FUN! BYE!] STOP]
   [CYCLE.Q]
END
```

We can represent the full program with the box chart below:



Here is a sample run to see how the program goes:

```
?QUIZ
HOW'S YOUR MULTIPLICATION? I WILL QUIZ
YOU ON THAT. JUST ENTER AN ANSWER TO
EACH QUESTION, AND YOU'LL BE TOLD IF
YOU'RE RIGHT OR NOT. I WILL KEEP A
RUNNING SCORE
WHAT IS 4 TIMES 6?
24
THAT'S RIGHT!
YOU'VE 1 OUT OF 1 QUESTIONS RIGHT. THAT'S 100%
DO YOU WISH TO CONTINUE? < Y OR N >
Y
WHAT IS 7 TIMES 8?
54
NOPE, THE ANSWER WAS 56.
DO YOU WISH TO CONTINUE? < Y OR N >
N
THANKS FOR THE FUN! BYE!
```

12.4.1 Improvements?

Well, the program runs, and it does what it's supposed to do. But that doesn't have to be the end of the exercise. Of course, there are extensions we can make, but right now we'd like to see if there are improvements we can make in the current program. One simplification is to make the CONTINUE? procedure more general so that we can use it in future programs. To do this we have to take the recursive call to CYCLE.Q out of the procedure. We can make this change by altering CONTINUE? so that it outputs the user's decision to CYCLE.Q. Then the recursive call to CYCLE.Q can come at the end of CYCLE.Q (tail recursion).

That's easy enough to do. In fact, we can remake CONTINUE into a predicate procedure and have it output "TRUE or "FALSE. Here are the revised listings for CYCLE.Q and CONTINUE?:

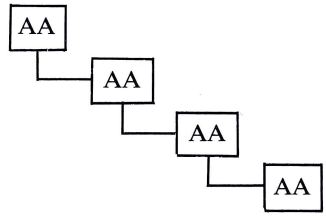
```
TO CYCLE.Q
FETCH
ASK
COMPARE
SCORE
IF CONTINUE? = "FALSE [PR [THANKS FOR THE FUN! BYE!] STOP]
```

```
CYCLE.Q
END


TO CONTINUE?
PR [DO YOU WISH TO CONTINUE? < Y or N >]
IF GETB.YN = "Y [OP "TRUE] [OP "FALSE]
END
```

What are the advantages to this change?
First, it makes the recursion more evident to someone reading the program. Now the call to CYCLE.Q comes at the end of CYCLE.Q itself instead of being hidden in another procedure. Also, we've created a new tool; we can use the CONTINUE? procedure in other programs, too. To the user, the new version will seem to run the same as the old, but the new version is more "programmer-friendly."

STOPPING RECURSIVE PROCEDURES

Recursive procedures like CYCLE.Q must always include a condition to stop them using either STOP, END, OUTPUT or THROW "TOPLEVEL. However, since a recursive procedure can create subprocedures at lower levels, we must be sure that *each level* also stops. To illustrate this, here is a box chart showing a typical recursive procedure creating lower levels



There are three types of recursive calls, each type requiring a different way to stop them. Let's first assume the procedure AA is type 1 recursion.

Type 1 . . . Tail End Recursion

This is the simplest type of recursion, one we've used extensively in graphics and list processing. We use an

```
IF (condition) [STOP]
```

like

```
IF :X > 20 [STOP]
```

placed *anywhere* in the procedure to stop the lowest level procedure. Each higher up level then ENDs. The CYCLE.Q procedure is an example.

Now let's assume that AA is type 2 recursion.

Type 2 . . . Recursion with OUTPUT

These procedures stop with the OUTPUT primitive. Often, there are two OUTPUTs in the procedure like this:

```
IF (condition) OUTPUT (something)
                      OUTPUT (recursive call)
```

Here is an example from Chapter 9

```
TO ADDUP :X
IF :X = 1 [OUTPUT 1]
OUTPUT :X + ADDUP :X - 1
END
```

The lowest level procedure stops with the IF-condition-OUTPUT line, then each higher up level stops with the OUTPUT-recursive call line. Each level must stop in order to pass information up to the next level.

Now let's assume that AA is type 3 recursion.

Type 3 . . . Complex Recursion

Complex recursive procedures are those that have some action taking place after the recursive call. A symbolic complex procedure would look like this.

```
TO AA
some action
IF (condition) [AA]
some more action
END
```

The COMPLEX procedure from Chapter 6 worked this way. The "action" parts of the program were to print the input and to stop the procedure after the input reached some value. Now if we place an IF (condition) STOP in the procedure, it will stop the lowest level procedure only. Each higher up level will then do "some more action" before ENDing. If this is what you want, okay. But if you want to stop *all* levels with the (condition), then the command would be:

```
IF (condition) [THROW "TOPLEVEL]
```

The command THROW "TOPLEVEL stops the lowest level procedure as well as all higher levels and takes you out of the procedure AA and gives you back the Logo prompt.

In general, we would suggest that beginning programmers avoid complex recursive procedures until you have a good idea of how these procedures work.

This doesn't exhaust the possibilities for improvement to our QUIZ program. In the next section, we will come up with some more ideas.

12.5 Programming Aids

Earlier we mentioned three programming aids: top-down programming, using tools, and documentation. How do they fit in with what we have been doing? The answer is that they fit in pretty well. In fact, we made use of all three while designing the QUIZ program. Let's see how.

First, the main idea of top-down programming is to take an overview of the whole problem and subdivide it into its main parts. Then we look at each part and, if necessary, divide it into smaller tasks yet. Then, when the tasks become small enough to be pretty simple, we write a separate procedure for each task. This is exactly the process we went through; we just didn't tell you that it was top-down programming. Top-down programming breaks a program up into separate modules. Logo's procedure-oriented approach makes it easy to use top-down design. Just use a separate procedure for each basic task.

Top-down programming has many advantages. It makes programming much easier, since each module is short. It also makes the program easier to test since we can test each module by itself and see if it works properly or not. And it makes the program easier to fix or alter. Just toss out the offending module and replace it with a better one. For example, suppose we want to give the user a chance to change his mind about his answer. Then we just replace the GETB.NUMBER procedure with the GETC.NUMBER procedure.

Secondly, we invoked the principle of using tools when we used GETB.NUMBER and GETB.YN. We even developed a new tool, CONTINUE?. Did we miss any other opportunities to develop a tool? Let's check over QUIZ and see if there are some procedures we could use in other programs. Hmm, SCORE seems like a good candidate. It could be used in any other quiz-like programs we might make. The only problem is that it is tied to the particular variables ("QUESTIONS and "CORRECT) used by QUIZ. We can fix that by remaking SCORE into a procedure with inputs:

```
TO SCORE :RIGHT :TRIES
PR (SE [YOU'VE] :RIGHT [OUT OF] :TRIES [RIGHT. THAT'S]
100 * :RIGHT / :TRIES [/ % .])
END
```

Of course, then we would have to change the call in CYCLE.Q to

```
SCORE :CORRECT :QUESTION
```

Another important point is that it is better to use local variables (such as procedure inputs) than global variables for a procedure you intend to use as a tool (We could also use the primitive LOCAL to do this). This helps in two ways. First, you can use the same procedure for a variety of variables. If some other program uses ":GOOD and ":ATTEMPTS for variable names, just use the call

```
SCORE :GOOD :ATTEMPTS
```

Secondly, keeping a variable local to a procedure prevents the procedure from accidentally changing a variable by the same name in another procedure.

That takes care of the first two programming aids. What did we do about documentation? In Logo we can do part of the job just by breaking the program into procedures and choosing descriptive names for them. For example, our procedure CYCLE.Q now looks like this (including the new SCORE):

```
TO CYCLE.Q
FETCH
ASK
COMPARE
SCORE :CORRECT :QUESTIONS
IF CONTINUE? = "FALSE [PR [THANKS FOR THE FUN! BYE!] STOP]
CYCLE.Q
END
```

This almost explains itself, but by looking at the names you might wonder, fetch what? ask what? Clearly, we could do more to document the program than we have. Logo does not have a REMARK facility like BASIC does, but we can create one easily. Just define a REMARK procedure this way:

```
TO REMARK :LIST
END
```

It may look to you like this procedure does nothing. Well, that's exactly what it is supposed to do! We can use it in CYCLE.Q this way:

```
TO CYCLE.Q
FETCH
REMARK [FETCHES 2 NUMBERS AND THEIR PRODUCT]
```

```
ASK
REMARK [ASKS THE USER WHAT THE PRODUCT OF THE 2 NUMBER IS]
COMPARE
SCORE :CORRECT :QUESTIONS
IF CONTINUE? = "FALSE [PR [THANKS FOR THE FUN! BYE!] STOP]
CYCLE.Q
END
```

You can put in as many clarifying remarks as you like. Since the REMARK function does nothing, they don't affect the running of your program (other than to slow it slightly and take room in the workspace); but their presence helps explain the workings of the program.

There are two kinds of documentation you can give programs—REMARKS to go inside the program or Box Chart information to go outside the program. Both kinds are helpful additions for the programmer.

Now let's return to the subject of keeping the reader happy and show how to improve the appearance of a program's output.

12.6 Screen Formatting Utilities

Here we will present two simple utilities to help you control the appearance of a program's output. The first lets you place spaces in the output; we call the utility SP. It looks like this:

```
TO SP :X
REPEAT :X [TYPE CHAR 32]
END
```

TYPE is like PRINT except that it does not move the cursor to a new line when it finishes. CHAR 32 is just the code for the space character (See Appendix B). Using SP, we can produce output like this:

```
?ECHOECHO
WHAT'S YOUR NAME?
FLOYD!
 FLOYD!
  FLOYD!
   FLOYD!
    FLOYD!
     FLOYD!
```

Listing

```
TO ECHOECHO
PRINT [WHAT'S YOUR NAME?]
MAKE "ANS SE READLIST [!]
SPACEOUT :ANS 5
END


TO SPACEOUT :X :N
IF :N = 0 [STOP]
SPACEOUT :X :N-1
SP :N
PRINT :X
END
```

Recall that placing the action commands before the recursive call gives a countdown effect (starting with the input N = 5, then 4 3 2 1), while placing the action commands after the recursive call gives a countup effect (N = 1 2 3 4 5). That is, level 5 spaceout prints its input, then level 4, etc., back up to the toplevel, which inserts 5 spaces. (There are *easier* ways to do the same thing; we just wanted to throw in a reminder of complex recursion.)

Our second utility is just a slight rephrasing of the Logo primitive SETCURSOR. This primitive, as you probably have guessed, sets the position of the cursor on the screen. It takes as input a list of two positive integers. The first number, which can be from 0 to 39, indicates the column. The columns are numbered from left to right. The second number, which can be from 0 to 23, describes which row. These are numbered from top to bottom. Our rephrasing is called SC, and this is its listing:

```
TO SC :X :Y
SETCURSOR SE :X :Y
END
```

As you can see, the only difference is that SC takes two inputs (each of which is a single number), while SETCURSOR takes one input, a list of two numbers. Thus, these two calls have the same effect:

```
SC 10 15
SETCURSOR [10 15]
```

Both place the cursor on the 11th column of the 16th row. (Notice that the numbering starts with 0, so 0 is the first column, and so on.)

We created SC because it is often used and faster to type. SC is like an abbreviation for SETCURSOR.

Let's use these utilities to modify the INIT.Q procedure of QUIZ.

12.7 Adding Formatting to Quiz

Our first version of INIT.Q looked like this:

```
TO INIT.Q
PR [HOW'S YOUR MULTIPLICATION? I WILL QUIZ]
PR [YOU ON THAT. JUST ENTER AN ANSWER TO]
PR [EACH QUESTION, AND YOU'LL BE TOLD IF]
PR [YOU'RE RIGHT OR NOT. I WILL KEEP A]
PR [RUNNING SCORE.]
MAKE "QUESTIONS 0
MAKE "CORRECT 0
END
```

Its printed output looked like this:

```
HOW'S YOUR MULTIPLICATION? I WILL QUIZ
YOU ON THAT. JUST ENTER AN ANSWER TO
EACH QUESTION, AND YOU'LL BE TOLD IF
YOU'RE RIGHT OR NOT. I WILL KEEP A
RUNNING SCORE.
```

Let's make the output look like this:

```
                    !!MATH QUIZ!!

              HOW'S YOUR MULTIPLICATION?

              I WILL QUIZ YOU ON THAT.

JUST ENTER AN ANSWER TO EACH QUESTION
AND YOU'LL BE TOLD IF YOU'RE RIGHT
OR NOT. I WILL KEEP A RUNNING SCORE.
```

It's easily done using our new utilities:

Listing

```
TO INIT.Q
CLEARTEXT
SC 2 13
PRINT [!!MATH QUIZ!!]
PRINT [ ]
SP 5 PR [HOW'S YOUR MULTIPLICATION?] PR [ ]
SP 5 PR [I WILL QUIZ YOU ON THAT.] PR [ ]
PR [JUST ENTER AN ANSWER TO EACH QUESTION]
PR [AND YOU'LL BE TOLD IF YOU'RE RIGHT]
PR [OR NOT.] PR [ ]
SP 5 PR [I WILL KEEP A RUNNING SCORE.]
MAKE "QUESTIONS 0
MAKE "CORRECT 0
END
```

Now we are beginning to have a good user-friendly interactive quiz program.

There are still many additions that could be made to it. Here are some of them:

1) Let the user choose the level of complexity of the problems.
2) Add in addition, subtraction and division quizzes.
3) Another refinement to QUIZ would be to start off with an upper limit of, say, five, to the numbers chosen to be multiplied. Then, if the user does well, this upper limit can be increased. To be more specific, once the first five turns have gone by, increase the upper limit by one every turn that the user's score is 80% or greater. [Hint: Create a new variable :LIMIT in INIT.Q and make it equal to five there. Use :LIMIT instead of 10 in FETCH. Then, in CYCLE.Q, insert a procedure to update the value of :LIMIT according to the rules outlined above.)

12.8 Summary

Humanizing a program means to make the program easy for someone to use and to make it easy for you or another programmer to follow. We suggest you use the following tips for making your programs friendly to user and writer alike.

User aids

1. Make your programs interactive
2. Use error trapping to head off possible user errors
3. Format the program output so that it is easy to read

Programmer Aids

1. Use top-down programming
2. Create short procedures as tools
3. Document your programs

12.9 Review Questions and Exercises

Review Questions

1. What are the major advantages of an interactive program?
2. What kind of data object does READLIST output?
3. Why do we often use the two primitives FIRST and READLIST together?
4. Give an example of a "positive" error trap. A "negative" error trap.
5. In the utility, GETB.NUMBER, why do we have line 3 read:

   MAKE "ANS READLIST

   Why don't we use:

   MAKE "ANS FIRST READLIST

   since we're looking for a number input?
6. Why does the utility GETB.NUMBER use "OUTPUT GETB.NUMBER", when making the recursive call? Why not leave the OUTPUT out and just call GETB.NUMBER? (The same is true for GETB.YN and GETC.NUMBER)
7. Why is the last listing for CYCLE.Q more readable than the first one?
8. Does the REMARK procedure always have to take a list as input?

Exercises at the Terminal

1. Create the utilities shown in Fig. 12-2.
2. Try making some of the changes suggested for QUIZ.
3. Humanize some of your favorite graphics programs.

Answers to Review Questions

(1) Friendly, less chance of "bombing" (2) A list (3) To remove a number or word from the list. (4) IF NOT MEMBERP :ANS [Y N] [OUTPUT GETB.NY], IF :ANS=[ ] [OUTPUT GETB.YN] (5) To allow the user to enter the empty list as input so we can trap it. (6) A recursive operation must use OUTPUT GETB.NUMBER in order to pass the answer up to each higher level. See the Box titled STOPPING RECURSIVE PROCEDURES. (7) The recursive call is easier to see. (8) No

CHAPTER 13

TWO PROGRAMS: MAKEGRAPH AND STORE.DATA

CONCEPTS:

graphing
using a disk file of data
modular programming

PRIMITIVES:

PACKAGE, PKGALL, BURY, UNBURY

13.0 Introduction

Now that we can write intelligent, humanized programs, it's time to get more ambitious with our programming projects. In Chapter 7 we promised you a graphing package, so let's work on that. After all, graphs are pretty useful. They are great for summarizing data like stock prices, rainfall records, your personal running records, and the like. Logo, with its ability to handle figures, lists, and graphics, is an excellent language for this purpose.

As usual, our first step is to decide what the package should do. A graph involves plotting one quantity (such as rainfall) against a second quantity (such as month), so we need to decide on some systematic way of representing such data. Note that a point on a graph corresponds to two numbers (to the rainfall and month, in our example). Thus a natural way to represent a point is with a list of two members. The first member will be the X-value (plotted along the horizontal axis), and the second member will be the Y-

value (plotted along the vertical axis). Thus, each such list represents a single point on the graph. Then we will use a list of lists to represent all the points. For example, this list of lists could represent rainfall for the first 6 months of the year:

```
[ [1 10] [2 7] [3 8] [4 2] [5 1] [6 0] ]
```

January (month 1) had 10 inches of rain, February had 7 inches, and so on. We will use this form later to store data. However, for the actual graphing process it is more convenient to use two separate lists, one for the X-values, and one for the Y-values, as we did in Chapter 7. Using this approach, the same data would be represented this way:

X-list: [1 2 3 4 5 6]
Y-list: [10 7 8 2 1 0]

This is the representation our new graphing program will use.

Next, we need to have the program read the data. Let's offer two choices: the user can enter the numbers interactively from the keyboard, or the user can ask the program to read the numbers from a disk file. (This raises a side question: how do we get data into a disk file in the first place? That is another programming problem, and we will handle it in this chapter, too.)

Once all the data is collected, it is a good idea to show the data to the user and let her decide if she wants to go ahead or not with the graphing.

If she does, the next major step is setting up the graph. Since we don't know in advance what range of numbers will be read in, it would be useful if the program would look at the numbers it gets and use them to decide what sort of graphing scale to use. This is called *autoscaling* and guarantees that the graph will neither run off the screen nor be scrunched down into one corner.

Finally, once the scale is set up, the points themselves should be plotted. Fig. 13-1 shows our first box chart for this program.

We can use a master procedure to set out these main points, and that is what we do next.

13.1 MAKEGRAPH: The Program Backbone

Here is our first listing:

Listing

```
TO MAKEGRAPH
```

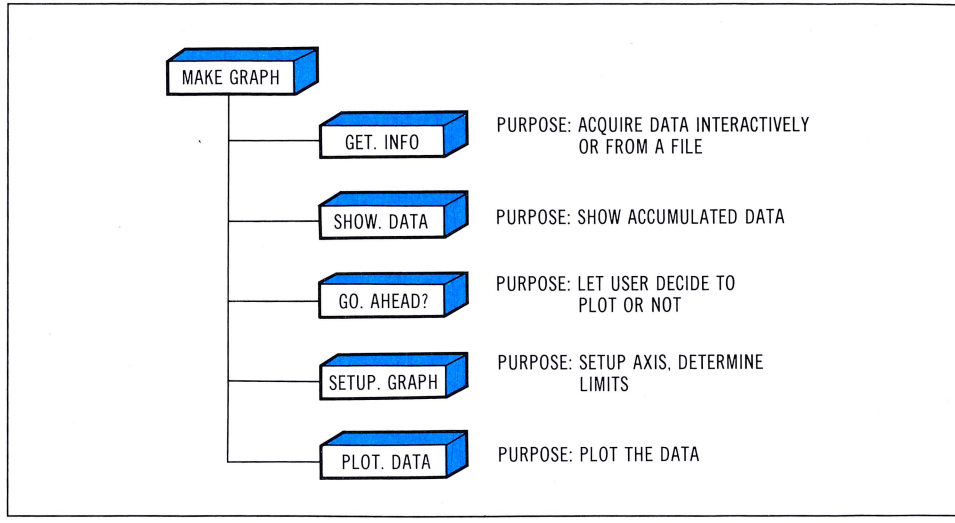


Fig. 13-1. Initial box chart for MAKEGRAPH

```
(LOCAL "XLIST "YLIST "XMIN "YMIN "XRANGE "YRANGE)
GET.INFO
SHOW.DATA
IF NOT GO.AHEAD? [PR [OK, BYE!] STOP]
SETUP.GRAPH
PLOT.DATA :XLIST :YLIST
END
```

Let's see what this procedure does. The LOCAL statement sets up local variables used by more than one procedure in the program. :XLIST and :YLIST will be used to store the X and Y values for the data. The other variables are used in setting up the boundaries of the graph. Next, GET.INFO gets the data for the graph. We'll get to the details later. Then SHOW.DATA prints out the data on the screen. GO.AHEAD? is a user-defined predicate that has the value "TRUE if the user wants to continue. The program is halted if the user does not want to go ahead. Next, SETUP.GRAPH sets up the axes for the graph and determines plotting scales that will be used. Finally, PLOT.DATA plots the points. This procedure works recursively, reducing its lists of data by one data pair each time it plots its points. That's why we use it with input lists, so that each recursive call uses new, shorter lists, until all the points are plotted and the lists are empty.

That's the overview. Now let's look more deeply. We begin, of course, with GET.INFO.

13.2 GET.INFO: Getting Numbers for the Graph

This is the most complex of the procedures in MAKEGRAPH, for it has to determine whether the user wants to supply the data interactively or from a file. Then it has to be able to cope with either choice. Here is the listing:

Listing

```
TO GET.INFO
LOCAL "MENU
RECYCLE
CS TEXTSCREEN CLEARTEXT SETCURSOR [0 0]
PR [HOW DO YOU WISH TO INPUT DATA?]
PR []
PR [1) INTERACTIVELY?]
PR [2) FROM A DISK FILE?]
PR []
PR [ENTER THE NUMBER OF YOUR CHOICE.]
MAKE "MENU GETB.12
MAKE "XLIST []
MAKE "YLIST []
IF :MENU = 1 [ENTER.DATA]
IF :MENU = 2 [LOAD.LIST]
END
```

Actually, this doesn't look so bad. Of course, we have hidden a lot of the work in additional procedures. Let's see what is going on here. First, the procedure clears the screen, puts the cursor in the upper left corner, and asks the user to choose his mode of entering data. Here we use a procedure GETB.12 to get the response. As you will see below, GETB.12 works like GETB.NUMBER, but it will only accept a 1 or a 2. This is an important bit of error checking, for the rest of the program assumes that those are the only possible values for :MENU. The two variables :XLIST and :YLIST are initialized to empty lists. Then the program runs the ENTER.DATA procedure if you wish to enter the data yourself, or the LOAD.LIST procedure if you want to read data from a disk.

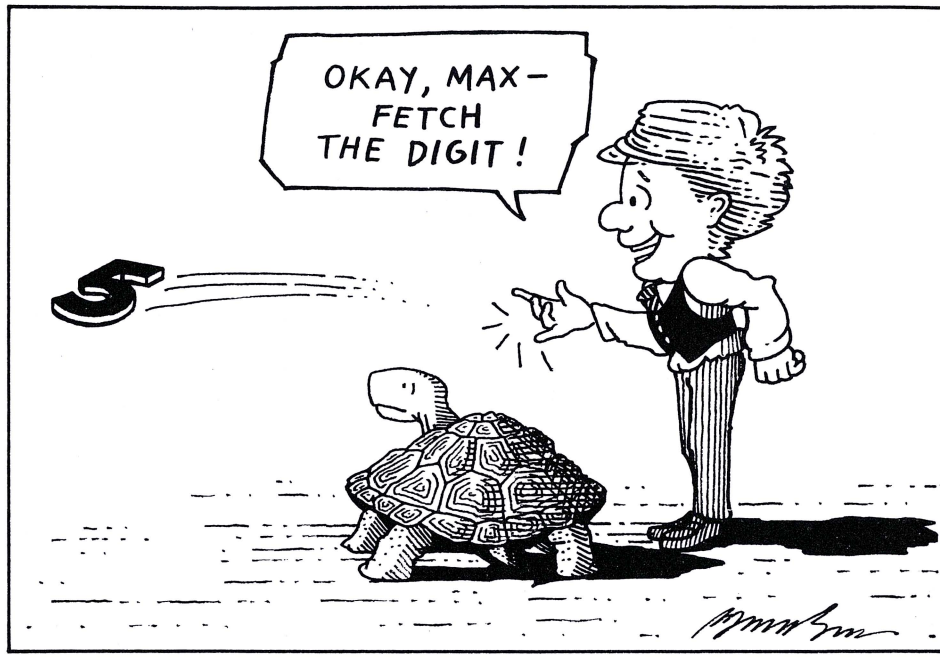
Next, let's turn to some of the new procedures used in GET.INFO. First, let's see GETB.12.

Listing

```
TO GETB.12
```

```
LOCAL "ANS
MAKE "ANS RL
IF :ANS = [] [PR [ENTER A 1 OR 2, PLEASE] OP GETB.12]
IF NOT MEMBERP FIRST :ANS [1 2] [PR [A 1 OR 2, PLEASE] OP
 GETB.12]
OUTPUT FIRST :ANS
END
```

This procedure is very much like GETB.NUMBER, except it limits input to just one or two instead of to any number. Like GETB.NUMBER, it rejects erroneous responses and keeps repeating via recursion until a satisfactory answer is received.



Incidentally, if you want to check if the program is working, you can set up dummy procedures for ENTER.DATA, LOAD.LIST, and the rest, and then run MAKEGRAPH. For example, you can define ENTER.DATA this way:

Listing (dummy procedure)

```
TO ENTER.DATA
```

```
PR [HEY! I'M IN ENTER.DATA!]
END
```

Once MAKEGRAPH works to your liking (you may, for example, wish to change the messages or change the method for following up on incorrect responses), you can turn to the task of writing the working versions of ENTER.DATA and its companions. Indeed, let's take up ENTER.DATA now.

We want this procedure to instruct the user how to enter a data pair for a graph point. The program should check to see that the user inputs two items and that both items are numbers. Then the first number should be added to the X-list and the second number added to the Y-list. Also, there should be a way for the user to tell the program when she is finished entering data. Fig. 13-2 gives the ENTER.DATA box chart.

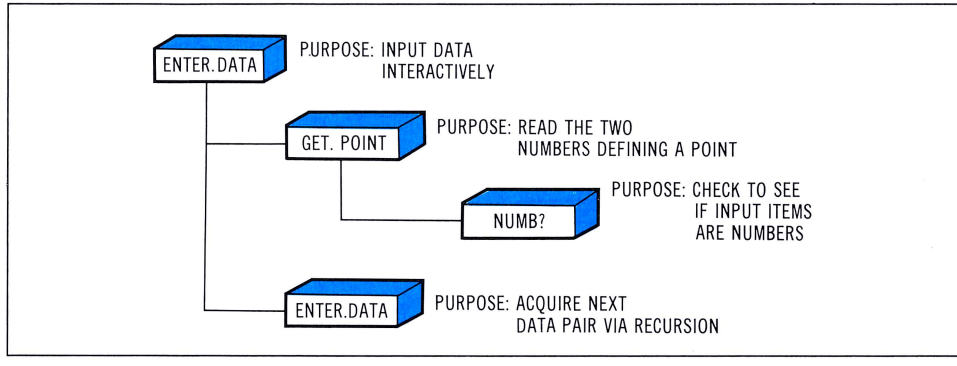

ENTER.DATA
PURPOSE: INPUT DATA INTERACTIVELY
GET. POINT
PURPOSE: READ THE TWO NUMBERS DEFINING A POINT
NUMB?
PURPOSE: CHECK TO SEE IF INPUT ITEMS ARE NUMBERS
ENTER.DATA
PURPOSE: ACQUIRE NEXT DATA PAIR VIA RECURSION

Fig. 13-2. Box chart for ENTER.DATA.

Our version will utilize two new procedures: GET.POINT and NUMB?. We'll list those, too. Here we go!

Listings

```
TO ENTER.DATA
LOCAL "TEMP
MAKE "TEMP []
PR []
PR [PLEASE ENTER ONE X Y PAIR, FOR]
PR [EXAMPLE, 2 5 OR 9 3]
PR []
PR [TO QUIT, HIT <RETURN> WITH NO DATA]
```

```
MAKE "TEMP GET.POINT
IF :TEMP = [] [STOP]
MAKE "XLIST LPUT FIRST :TEMP :XLIST
MAKE "YLIST LPUT LAST :TEMP :YLIST
PR [PLEASE CONTINUE]
ENTER.DATA
END

TO GET.POINT
LOCAL "ANS
MAKE "ANS RL
IF :ANS = [] [OUTPUT :ANS]
IF NOT (COUNT :ANS) = 2 [PR [I NEED 2 INPUTS. TRY AGAIN] OP
   GET.POINT]
IF (NUMB? :ANS) = "FALSE [PR [NUMBERS ONLY. TRY AGAIN] OP
   GET.POINT]
(PR [YOU HAVE ENTERED THE PAIR] :ANS)
PR [IS THIS WHAT YOU WANT? Y OR N?]
IF GETB.YN = "N [PR [TRY AGAIN] OP GET.POINT]
OUTPUT :ANS
END

TO NUMB? :LIST
TEST AND NUMBERP FIRST :LIST NUMBERP LAST :LIST
IFT [OUTPUT "TRUE]
IFF [OUTPUT "FALSE]
END
```

This procedure is pretty straightforward. We tell the user to enter two numbers for each graph point and to hit a simple RETURN to end the process. The GET.POINT procedure checks to make sure that only these two possibilities (two numbers or else the empty list) get through. GET.POINT uses NUMB? to see if the entries are, indeed, numbers. Next, GET.POINT gives the user the opportunity to correct an entry data. Then ENTER.DATA assigns the first member of the pair to the X-list, and it puts the second member into the Y-list. Tail recursion then causes the process to be repeated.

What next? The next procedure in GET.INFO is LOAD.LIST, and that deals with getting data from a file. Let's postpone that part for the while and move on to the graphing part. Later we can come back to deal with the disk file option.

13.3 SHOW.DATA and GO.AHEAD?

First, we remind the user what data is present:

Listing

```
TO SHOW.DATA
PR [HERE ARE THE DATA:]
PR SE [X LIST] :XLIST
PR SE [Y LIST] :YLIST
END
```

We simply print the two lists. Next, we had the line

```
IF NOT GO.AHEAD? [PR [OKAY, BYE!] STOP]
```

and this is what GO.AHEAD looks like:

Listing

```
TO GO.AHEAD?
PR [SHALL WE GRAPH? Y OR N?]
MAKE "ANS GETB.YN
IF :ANS = "Y [OP "TRUE] [OP "FALSE
END
```

Now let's see how to set up the graph.

13.4 SETUP.GRAPH: Setting Up the Graph

We will give these duties to SETUP.GRAPH:

1. Draw an X-axis along the bottom of the display area, and draw a Y-axis along the left-hand side.
2. Determine the largest and smallest values of X and Y.
3. Use this information to choose graphing scales so that the graph fills the screen.

Some of these duties, of course, will be passed on to procedures called by SETUP.GRAPH. This makes SETUP.GRAPH itself look pretty simple:

Listing

```
TO SETUP.GRAPH
RECYCLE
CLEARTEXT
DRAWAXIS
MAKE "YMIN LO :YLIST
MAKE "XMIN LO :XLIST
MAKE "YRANGE (HI :YLIST) - :YMIN
MAKE "XRANGE (HI :XLIST) - :XMIN
END
```

HI - p. 271
LO - p. 272

The RECYCLE request clears away any garbage left over from the data acquisition phase. DRAWAXIS, as you may have guessed, draws the axes. It looks like this:

Listing

```
TO DRAWAXIS
HT
PU
SETX -120
PD FD 75 BK 150 PU
HOME
SETY -75
PD RT 90 FD 120 BK 240 PU
HOME
END
```

It draws a vertical axis along the left side (X = −120) of the work area and a horizontal axis along the bottom (Y = −75).

Next, SETUP.GRAPH uses the HI and LO math functions of Chapter 10 to find the low values and the range of X and of Y. Now we are ready to plot, so let's move on to PLOT.DATA.

13.5 PLOT.DATA: Putting Points on the Screen

PLOT.DATA finishes the job with the help of MARK.XY. Let's look at those procedures now:

Listings

```
TO PLOT.DATA :XDATA :YDATA
(LOCAL "X "Y)
IF :XDATA = [] [STOP]
MAKE "X ((FIRST :XDATA) - :XMIN) * 240 / :XRANGE - 120
MAKE "Y ((FIRST :YDATA) - :YMIN) * 150 / :YRANGE - 75
MARK.XY :X :Y
PLOT.DATA BF :XDATA BF :YDATA
END

MARK.XY :X :Y
HT
PU
SETX :X
SETY :Y
PD
RT 45 FD 1 BK 2 FD 1
LT 90 FD 1 BK 2 FD 1
RT 45
PU
END
```

PLOT.DATA begins with the first value in the X-list and scales it. Notice that if :X equals :XMIN, then :X is given the value of −120, i.e., to the far left of the plotting area. If X has its largest value, then :X winds up assigned a value of 120, i.e., to the far right. Other values of X wind up between these two extremes. Similarly, :Y is assigned a value that places it somewhere between the bottom and top of the plotting area.

The MARK.XY procedure then places a small "X" at the corresponding point.

Then PLOT.DATA calls itself recursively, stripping the first data value from each list. The procedure halts when it empties the list. If you use a great number of points, you may wish to insert a RECYCLE statement into this procedure.

Fig. 13-3 summarizes the program to this point. (All that is lacking is the elucidation of LOAD. LIST.)

13.6 Trying It Out

Let's see how the program works. Here is a sample run in which we feed in a date for X and the US population in millions for Y:

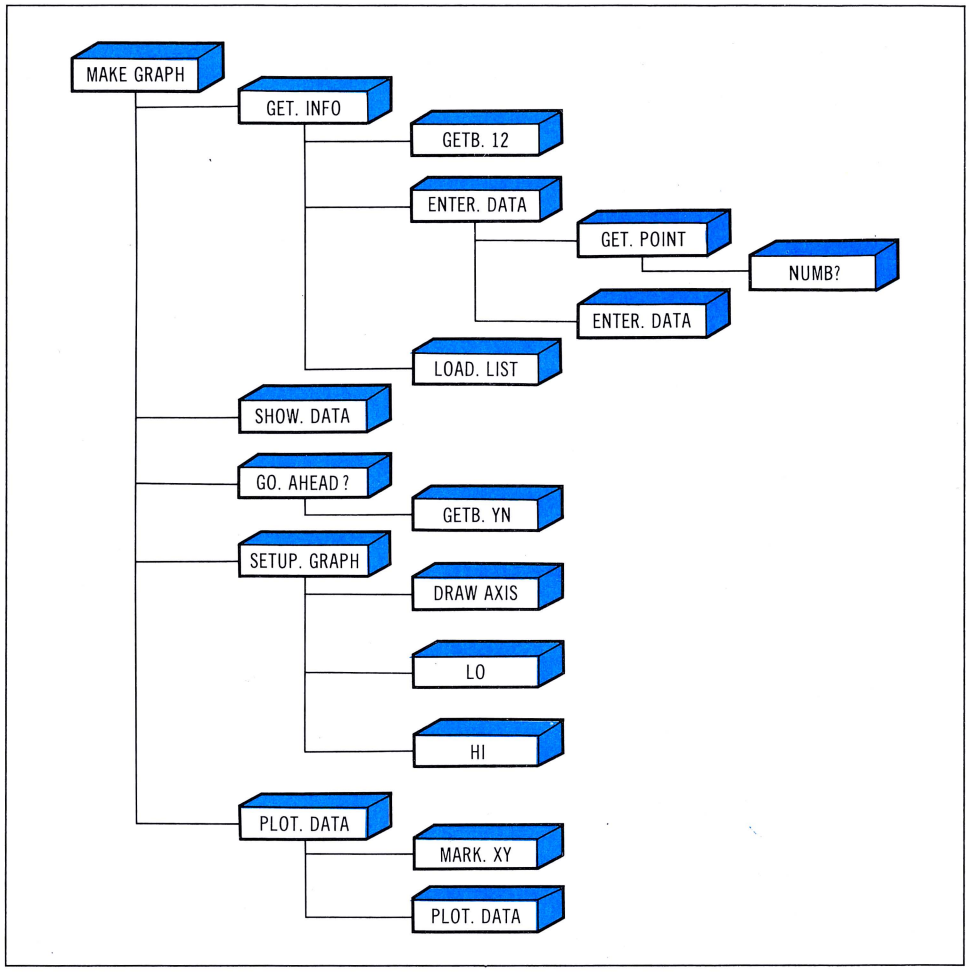


Fig. 13-3. Intermediate box chart.

Sample run

```
?MAKEGRAPH
HOW DO YOU WISH TO INPUT DATA?

1) INTERACTIVELY?
2) FROM A DISK FILE?

ENTER THE NUMBER OF YOUR CHOICE.
 1
```

```
PLEASE ENTER ONE X Y PAIR, FOR
EXAMPLE, 2 5 OR 3 9

TO QUIT, HIT <RETURN> WITH NO DATA
1790 3.9
YOU HAVE ENTERED THE PAIR 1790 3.9
IS THIS RIGHT? Y OR N?
Y
PLEASE CONTINUE.

PLEASE ENTER ONE X Y PAIR, FOR
EXAMPLE, 2 5 OR 3 9

TO QUIT, HIT <RETURN> WITH NO DATA
1820
I NEED 2 INPUTS. TRY AGAIN
I WILL
NUMBERS ONLY. TRY AGAIN
1820 7.2
YOU HAVE ENTERED THE PAIR 1820 7.2
IS THIS RIGHT Y OR N?
Y
PLEASE CONTINUE.

PLEASE ENTER ONE X Y PAIR, FOR
EXAMPLE, 2 5 OR 3 9

TO QUIT, HIT <RETURN> WITH NO DATA
1850 23.2
YOU HAVE ENTERED THE PAIR 1850 23.2
IS THIS RIGHT? Y OR N?
Y
PLEASE CONTINUE.

PLEASE ENTER ONE X Y PAIR, FOR
EXAMPLE, 2 5 OR 3 9

TO QUIT, HIT <RETURN> WITH NO DATA
1880 50.2
YOU HAVE ENTERED THE PAIR 1880 50.2
IS THIS RIGHT? Y OR N?
Y
PLEASE CONTINUE.
```

```
PLEASE ENTER ONE X Y PAIR, FOR
EXAMPLE, 2 5 OR 3 9

TO QUIT, HIT <RETURN> WITH NO DATA
1910 105.7
HAVE YOU ENTERED THE PAIR 1910 105.7
IS THIS RIGHT? Y OR N?
Y
PLEASE CONTINUE.

PLEASE ENTER ONE X Y PAIR, FOR
EXAMPLE, 2 5 OR 3 9

TO QUIT, HIT <RETURN> WITH NO DATA
1940 131.7
HAVE YOU ENTERED THE PAIR 1940 105.7
IS THIS RIGHT? Y OR N?
Y
PLEASE CONTINUE.

PLEASE ENTER ONE X Y PAIR, FOR
EXAMPLE, 2 5 OR 3 9

TO QUIT, HIT <RETURN> WITH NO DATA
1970 203.3
YOU HAVE ENTERED THE PAIR 1970 203.3
IS THIS RIGHT? Y OR N?
Y
PLEASE CONTINUE.

PLEASE ENTER ONE X Y PAIR, FOR
EXAMPLE, 2 5 OR 3 9

TO QUIT, HIT <RETURN> WITH NO DATA
1980 226.5
YOU HAVE ENTERED THE PAIR 1980 226.5
IS THIS RIGHT? Y OR N?
Y
PLEASE CONTINUE.

PLEASE ENTER ONE X Y PAIR, FOR
EXAMPLE, 2 5 OR 3 9

TO QUIT, HIT <RETURN> WITH NO DATA
```

```
<RETURN>
HERE ARE THE DATA:
X LIST 1790 1820 1850 1880 1910 1940 1970 1980
Y LIST 3.9 7.2 23.2 50.2 105.7 131.7 203.3 226.5
SHALL WE GRAPH? Y OR N?
Y
```

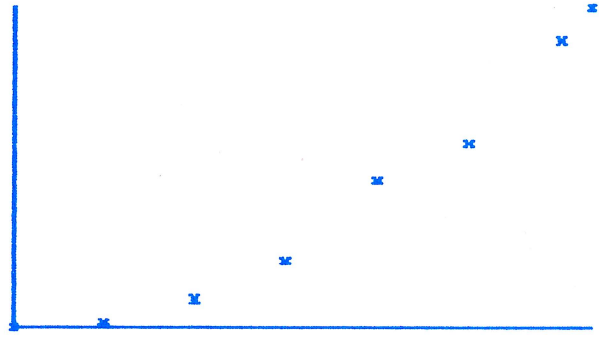


Fig. 13-4. The graph.

13.7 Graphing from a File

We've got the program to work interactively. Now let's see what is involved in reading data from a file. First, how can we get data into a file? We can assign a list of data to a variable, erase or bury everything else, and then use the SAVE command. This approach would look like this:

```
?ERPS
?ERNS
?MAKE "LIST [ [1790 3.9] [1820 7.2] [1850 23.2] ]
?SAVE "USPOP
```

The first two lines empty the workspace. The third line creates the new data, and the final line stores it. This will produce a disk file called "USPOP which contains the variable named :LIST. In a bit, we will show you a somewhat more general program for producing and adding to disk files, but for now we will concentrate on the fact that it produces the same sort of result that we have above.

Okay, suppose we have such a list of data on file. Then our MAKEGRAPH program will have to do two things:

1. Retrieve the contents of the file from the disk.
2. Transform them to the same form (an X-list and a Y-list) that the program uses.

These are the tasks that LOAD.LIST and its subsidiary procedures take on. Let's see how they do it.

Listing

```
TO LOAD.LIST
LOAD.FILE
MAKE.LISTS :LIST
END
```

Obviously LOAD.FILE fetches the data, and MAKE.LISTS constructs the X-list and the Y-list from the list of data pairs in the file.

Listing

```
TO LOAD.FILE
LOCAL "FILENAME
PR [WHAT IS THE NAME OF THE DATA FILE?]
MAKE "FILENAME GETB.WORD
LOAD :FILENAME
PR []
PR [THE DATA IN THE FILE ARE:]
PR :LIST
WAIT 240
END
```

Note that we have to provide the name of the file we want. (We have used the GETB.WORD utility for that purpose.) Also note that the program assumes that the name of the variable stored in the file is :LIST. So when we create a file to be used by this program, we can call the file what we like, but we must call the variable in the file :LIST. Bear in mind, that :LIST is a list of two-element lists, with each of these lists corresponding to a single graph point.

Now let's see how the data pairs in :LIST are divided up into two lists:

Listing

```
TO MAKE.LISTS :LIST
```

```
IF :LIST = [] [STOP]
MAKE "XLIST FPUT FIRST FIRST :LIST :XLIST
MAKE "YLIST FPUT LAST FIRST :LIST :YLIST
MAKE.LISTS BF :LIST
END
```

How does this work? Consider the example we gave earlier with

```
:LIST = [ [1790 3.9] [1820 7.2] [1850 23.2] ]
```

Then

```
FIRST :LIST = [1790 3.9]
```

i.e., the first data pair in a list of data pairs. It then follows that

```
FIRST FIRST :LIST = 1790
```

and

```
LAST FIRST :LIST = 3.9
```

The FPUT primitive then places these two values into :XLIST and :YLIST respectively. Then MAKE.LISTS calls itself recursively, stripping off the first data pair from :LIST. Now [1820 7.2] becomes the new first element, and the process continues until the list is emptied.

That does it. MAKEGRAPH works with data we type in interactively, and it works with data stored in data-pair form in a variable called :LIST stored in a disk file file. Our package is complete now, so let's show the whole program (aside from utilities from earlier chapters) collected together in Fig. 13-5. We indent procedures according to the level they are initially called. Wow! It's huge! Yet because we modularized the job, putting the program together was not overwhelming at all. Fig. 13-6 shows the organization with a box chart.

```
TO MAKEGRAPH
(LOCAL "XLIST "YLIST "XMIN "YMIN "XRANGE "YRANGE
GET.INFO
SHOW.DATA
IF NOT GO.AHEAD? [PR [OK, BYE!] STOP]
SETUP.GRAPH
PLOT.DATA :XLIST :YLIST
END
```

Fig. 13-5. MAKEGRAPH program.

```
TO GET.INFO
LOCAL "MENU
RECYCLE
CS TEXTSCREEN CLEARTEXT SETCURSOR [0 0]
PR [HOW DO YOU WISH TO INPUT DATA?]
PR []
PR [1) INTERACTIVELY?]
PR [2) FROM A DISK FILE?]
PR []
PR [ENTER THE NUMBER OF YOUR CHOICE.]
MAKE "MENU GETB.12
MAKE "XLIST []
MAKE "YLIST []
IF :MENU = 1 [ENTER.DATA]
IF :MENU = 2 [LOAD.LIST]
END

   TO GETB.12
   LOCAL "ANS
   MAKE "ANS RL
   IF :ANS = [] [PR [ENTER A 1 OR 2, PLEASE]
      OP GETB.12]
   IF NOT MEMBERP FIRST :ANS [1 2] [PR [A 1 OR 2,
      PLEASE] OP GETB.12]
   OUTPUT FIRST :ANS
   END

 TO ENTER.DATA
 LOCAL "TEMP
 MAKE "TEMP []
 PR []
 PR [PLEASE ENTER ONE X Y PAIR, FOR]
 PR [EXAMPLE, 2 5 OR 9 3]
 PR []
 PR [TO QUIT, HIT <RETURN> WITH NO DATA]
 MAKE "TEMP GET.POINT
 IF :TEMP = [] [STOP]
 MAKE "XLIST LPUT FIRST :TEMP :XLIST
 MAKE "YLIST LPUT LAST :TEMP :YLIST
 PR [PLEASE CONTINUE]
 ENTER.DATA
 END
```

Fig. 13-5.– cont. MAKEGRAPH program.

```
  TO GET.POINT
  LOCAL "ANS
  MAKE "ANS RL
  IF :ANS = [] [OUTPUT :ANS]
  IF NOT COUNT :ANS = 2 [PR [I NEED 2 INPUTS. TRY
     AGAIN] OP GET.POINT]
  IF NUMB? :ANS = "FALSE [PR [NUMBERS ONLY. TRY
     AGAIN] OP GET.POINT]
  (PR [YOU HAVE ENTERED THE PAIR] :ANS)
  PR [IS THIS WHAT YOU WANT? Y OR N?]
  IF GETB.YN = "N [PR [TRY AGAIN] OP
     GET.POINT]
  OUTPUT :ANS
  END

  TO NUMB? :LIST
  TEST AND NUMBERP FIRST :LIST NUMBERP LAST :LIST
  IFT [OUTPUT "TRUE]
  IFF [OUTPUT "FALSE]
  END

TO LOAD.LIST
LOAD.FILE
MAKE.LISTS :LIST
END

   TO LOAD.FILE
   LOCAL "FILENAME
   PR [WHAT IS THE NAME OF THE DATA FILE?]
   MAKE "FILENAME GETB.WORD
   LOAD :FILENAME
   PR []
   PR [THE DATA IN THE FILE ARE:]
   PR :LIST
   WAIT 240
   END

   TO MAKE.LISTS :LIST
   IF :LIST = [] [STOP]
   MAKE "XLIST FPUT FIRST FIRST :LIST :XLIST
   MAKE "YLIST FPUT LAST FIRST :LIST :YLIST
   MAKE.LISTS BF :LIST
   END
```

Fig. 13-5.– cont. MAKEGRAPH program.

```
TO SHOW.DATA
PR [HERE ARE THE DATA:]
PR SE [X LIST] :XLIST
PR SE [Y LIST] :YLIST
END

TO GO.AHEAD?
PR [SHALL WE GRAPH? Y OR N?]
MAKE "ANS GETB.YN
IF :ANS = "Y [OP "TRUE] [OP "FALSE
END

TO SETUP.GRAPH
RECYCLE
CLEARTEXT
DRAWAXIS
MAKE "YMIN LO :YLIST
MAKE "XMIN LO :XLIST
MAKE "YRANGE (HI :YLIST) - :YMIN
MAKE "XRANGE (HI :XLIST) - :XMIN
END

   TO DRAWAXIS
   HT
   PU
   SETX -120
   PD FD 75 BK 150 PU
   HOME
   SETY -75
   PD RT 90 FD 120 BK 240 PU
   HOME
   END

TO PLOT.DATA :XDATA :YDATA
(LOCAL "X "Y)
IF :XDATA = [] [STOP]
MAKE "X ((FIRST :XDATA) - :XMIN) * 240 / :XRANGE -120
MAKE "Y ((FIRST :YDATA) - :YMIN) * 150 / :YRANGE -75
MARK.XY :X :Y
PLOT.DATA BF :XDATA BF :YDATA
END

   MARK.XY :X :Y
```

Fig. 13-5.– cont. MAKEGRAPH program.

```
HT
PU
SETX :X
SETY :Y
PD
RT 45 FD 1 BK 2 FD 1
LT 90 FD 1 BK 2 FD 1
RT 45
PU
END
```

Fig. 13-5.–cont. MAKEGRAPH program.

13.8 STORE.DATA: Saving a List on Disk

It is very nice and useful to be able to store data on a disk. It is even nicer and more useful if we can update the data. If, for example, we are following the progress of a stock, we could add new data daily to a file containing the stock's value. Then, say, each week we could have MAKEGRAPH read that file and generate a graph of those prices.

Already we have shown a simple means to store a file on disk. Now we will present a program that stores data on disk. To make the program flexible, we will offer two types of lists for storage. The first kind of list is a list of data pairs, like the one we used above in MAKEGRAPH. The second kind of list is a list of single elements, such as a word list. In addition the program can create a new file or add to an existing file.

These elaborations complicate the program, but the basic idea remains the same as in the previous section: create the desired list, bury or erase everything else, then use the SAVE primitive. Here is the master procedure:

Listing

```
TO STORE.DATA
(LOCAL "MENU "FILE.NO "FILENAME)
INIT
GET.DATA
SAVE.DATA
END
```

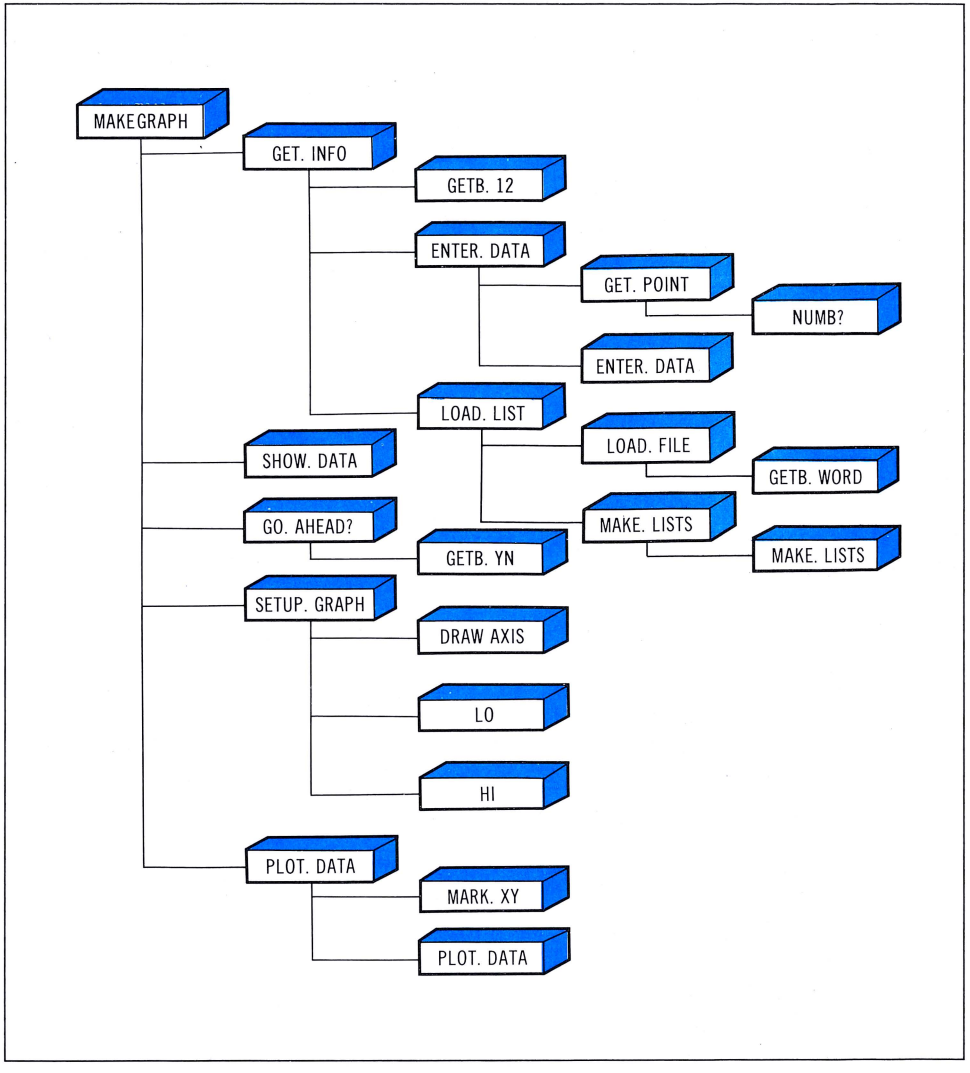


Fig. 13-6. The MAKEGRAPH box chart.

As we will see, INIT tells the user what the program does. It also finds out what kind of file the user wishes to create. Then GET.DATA takes over. It finds out whether the user wishes to create a new file or to add to an old one, and then it gathers the data interactively. Finally, SAVE.DATA buries the program and saves the data. We'll look at each procedure in turn. Fig. 13-7 summarizes these activities in a box chart.

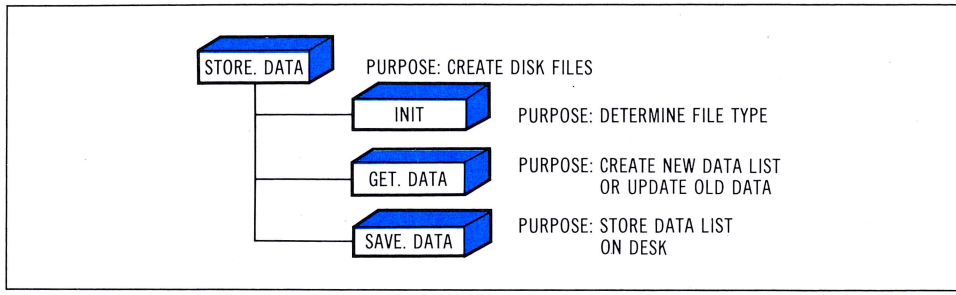


Fig. 13-7. SAVE.DATA box chart.

13.9 INIT: Starting Out

We just outlined what INIT does. Here are the details:

Listing

```
TO INIT
CLEARTEXT RECYCLE
PR [STORE.DATA CREATES OR ADDS TO FILES]
PR [THAT WILL BE STORED ON DISK. IT CAN]
PR [MAKE 2 DIFFERENT TYPES OF FILES:]
PR []
PR [1) A LIST OF NUMBER PAIRS, EG, X-Y]
PR [VALUES FOR GRAPH PLOTTING.]
PR []
PR [2) A LIST OF ELEMENTS, EG, NUMBERS]
PR [FOR A HISTOGRAM, OR WORDS FOR A LIST.]
PR []
PR [WHICH TYPE OF FILE DO YOU WANT? 1 OR 2?]
MAKE "FILE.NO GETB.12
END
```

Most of the procedure prints information, but it does use the procedure GETB.12 to fetch the user's answer. GETB.12 accepts a 1 or a 2 as a proper answer; we presented it earlier in this chapter.

Once INIT is finished, the procedure knows (through the value of :FILE.NO) what type of file the user wants. The next procedure, GET.DATA, uses this information.

13.10 GET.DATA: Data Input

This procedure finds out whether you wish to create a new file or add to an old one, finds the file name you want to use, then sees to the data gathering.

Listing

```
TO GET.DATA
CLEARTEXT
PR [WHICH DO YOU WANT TO DO?]
PR []
PR [1) CREATE A NEW FILE]
PR [2) ADD TO AN EXISTING FILE]
PR []
PR [PLEASE ENTER A 1 OR A 2]
MAKE "MENU GETB.12
GET.NAME
IF AND :FILE.NO = 1 :MENU = 1 [MAKE "LIST[] ADD.PTS]
IF AND :FILE.NO = 1 :MENU = 2 [ADD.PTS]
IF AND :FILE.NO = 2 :MENU = 1 [PR [TYPE IN YOUR LIST OF
   DATA] MAKE "LIST GETC.LIST]
IF AND :FILE.NO = 2 :MENU = 2 [ADD.LIST]
END
```

Recall our convention that an indented line is to be typed as a continuation of the preceding line. Here we use GETB.12 once again. GET.NAME is a new procedure that gets the file name. It looks like this:

Listing

```
TO GET.NAME
PR []
PR [WHAT IS THE FILE NAME?]
MAKE "FILENAME GETB.WORD
PR[]
IF :MENU = 1 [STOP]
LOAD :FILENAME
PR []
PR [HERE ARE THE EXISTING DATA:]
PR :LIST
END
```

Here we use the GETB.WORD procedure we developed earlier. If :MENU = 1 (creating a new file), the procedure stops after getting the name. Otherwise (adding to an old file) it loads the old file and prints its contents. Note that the program assumes that :LIST is the name of the variable stored in the file.

Once the program has all this information, it chooses one of four branches:

1. Data pairs, new file
2. Data pairs, old file
3. Data elements, new file
4. Data elements, old file

Fig. 13-8 illustrates this organization.

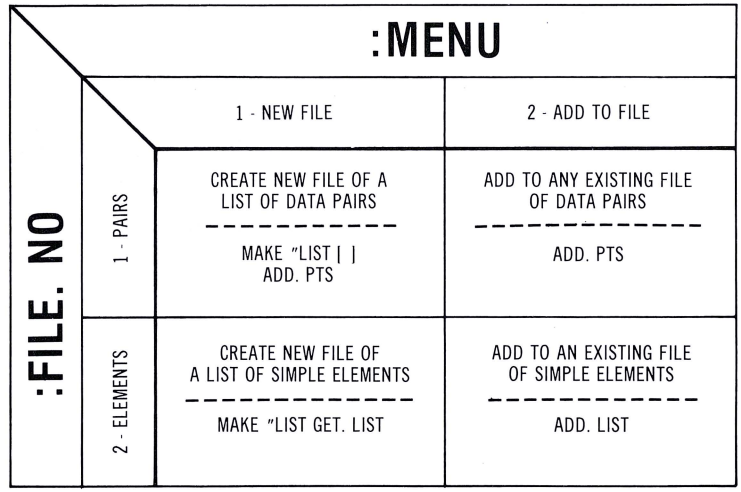

:MENU
1 - NEW FILE
2 - ADD TO FILE
:FILE. NO
1 - PAIRS
2 - ELEMENTS
CREATE NEW FILE OF A LIST OF DATA PAIRS
MAKE "LIST [ ]
ADD. PTS
ADD TO ANY EXISTING FILE OF DATA PAIRS
ADD. PTS
CREATE NEW FILE OF A LIST OF SIMPLE ELEMENTS
MAKE "LIST GET. LIST
ADD TO AN EXISTING FILE OF SIMPLE ELEMENTS
ADD. LIST

Fig. 13-8. GET.DATA organization.

The first two options both use the procedure ADD.PTS. The difference is that choice number 1 initializes :LIST to the null list ([ ]). For choice number 2, :LIST already has the contents loaded in from the named file. Let's look at ADD.PTS now.

Listing

```
TO ADD.PTS
LOCAL "ENTRY
```

```
PR []
PR [TYPE IN ONE (X,Y) DATA PAIR TO]
PR [BE ADDED.]
PR []
PR [TO QUIT ADDING, HIT A <RETURN>]
PR [WITH NO DATA.]
PR []
MAKE "ENTRY GET.POINT
IF :ENTRY = [] [STOP]
MAKE "LIST LPUT :ENTRY :LIST
PR :LIST
ADD.PTS
END
```

Note that we use the GET.POINT procedure we developed for MAKEGRAPH. Once two numbers are entered successfully, the two element list they form (:ENTRY) is added on to :LIST. For example, if :LIST is [[1 3] [2 6]] and you enter the numbers 4 and 9, then :ENTRY becomes the list [4 9], and :LIST becomes [[1 3] [2 6] [4 9]]. Then the current list is printed, and the process repeats until the user hits RETURN with no data.

That takes care of the two possibilities for the data pair files. What about the simple element files?

If the file is a new one (case 3), we use the utility GETC.LIST to get the list. Here the input list is read in in one fell swoop rather than element by element. GETC.LIST goes like this:

Listing

```
TO GETC.LIST
LOCAL "ANS
MAKE "ANS RL
IF :ANS = [] [PR [PLEASE ENTER A LIST, THEN <RETURN>]
   OP GETC.LIST]
PR []
PR [IS THIS RIGHT? Y OR N?]
IF GETB.YN = "Y [OP :ANS] [PR [] PR [ENTER NEW LIST]
   OP GETC.LIST]
END
```

Note that this gives the user a chance to change her mind. Also, it uses the utility GETB.YN to fetch the user's reply.

What about case 4, where we add to a file? We call upon ADD.LIST to

do the job. It prompts the user to type in the new list elements, uses GETC.LIST to read them in, then combines the new information with the old. Finally, it prints out the resulting list.

Listing

```
TO ADD.LIST
LOCAL "ENTRY
PR [TYPE IN ALL DATA ELEMENTS TO BE]
PR [ADDED TO THE EXISTING LIST.]
PR []
MAKE "ENTRY GETC.LIST
MAKE "LIST SE :LIST :ENTRY
PR [HERE IS THE COMPLETE LIST:
PR []
PR :LIST
WAIT 300
END
```

Fig. 13-9 is a box chart showing GET.DATA.

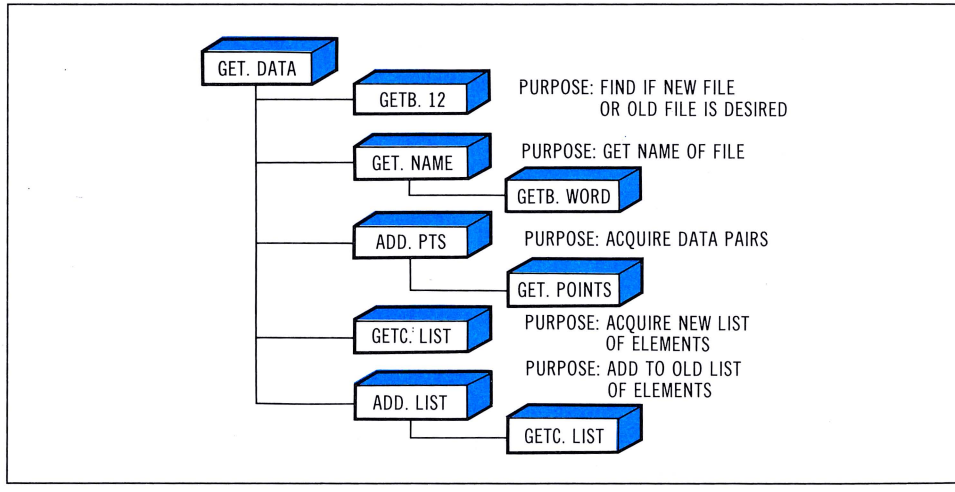


Fig. 13-9. Box chart for GET.DATA

Now that we've got the information we want stored in "LIST, we can save it.

13.11 SAVE.DATA: Saving the List

The basic tasks of this procedure are to clear the workspace of everything but :LIST and then to save :LIST. We have to clear out (or hide) the program or else the SAVE command will save the program along with the list on the disk file. To these basics we will add giving the user the option to back out.

Here is the listing:

Listing

```
TO SAVE.DATA
LOCAL "ANS
PACKAGE "STORE.DATA [STORE.DATA SAVE.DATA GET.NAME GET.DATA
   INIT ADD.PTS ADD.LIST GETB.YN GETC.LIST GETB.12
   GETB.WORD
BURY "STORE.DATA
PACKAGE "LI.ST [LIST]
PKGALL "RE.ST
BURY "RE.ST
CLEARTEXT
PR [ALL PROCEDURES ARE NOW TEMPORARILY BURIED]
PR []
(PR [I'M READY TO SAVE DATA IN] :FILENAME)
PR [IS THIS WHAT YOU WANT? Y OR N?]
MAKE "ANS GETB.YN
IF :ANS = "N [PR [OK, MAYBE SOME OTHER TIME.] STOP]
PR [HERE GOES!]
If :MENU = 2 [PR [GOT TO ERASE THE OLD FILE FIRST]
   ERASEFILE :FILENAME
SAVE :FILENAME
ERN "LIST
PR [DONE!]
UNBURY "STORE.DATA
UNBURY "RE.ST
PR [ALL PROCEDURES ARE RESTORED]
END
```

First, how did we handle the task of clearing the workspace? Our main tools are the Logo primitives PACKAGE, PKGALL, and BURY. The PACKAGE command lumps together several procedures into (what else?) a package. The command takes two inputs. The first is the name to be given

to the package, and the second is a name (or list of names) of the item (or items) to be placed in the package. In our procedure, for example, the name of the package is "STORE.DATA, and the second input is the list of all the procedures of the program. Note that we now have two "STORE.DATAs. One is a procedure, and the other is the name of a package. This doesn't confuse Logo since they are used differently. For example, if you say PO "STORE.DATA, Logo will look for the procedure, since PO takes a procedure (and not a package) for an input.

Suppose we have a package called "STORE.DATA. How does that help? It helps because we then can use the BURY command. This command takes a package name for an input, and it "buries" the package. This means it hides the package from certain commands, such as POALL, ERALL, and SAVE. A procedure in a buried package still runs when you type its name, so it is not gone; it's just partially invisible. (Incidentally, the UNBURY command will bring a package back. For instance, UNBURY "STORE. DATA would restore our program to the visible workspace, and we use that command at the end of this procedure.)

That takes care of hiding the program, but you might have some other procedures in your workspace. The PKGALL command helps with them. Any unpackaged names left in the workspace get bundled together under whatever name you give PKGALL as input. For instance, we use PKGALL "RE.ST to gather these remaining names together, and then we bury the package "RE.ST. (We chose the name "RE.ST in the hope it wouldn't conflict with any package names you had already.) Of course, this would have buried "LIST, too, so first we packaged :LIST to prevent it from being picked up by PGKALL.

Now we've taken care of nearly everything. Our program is buried in "STORE.DATA. :LIST sits in the workspace. Other stray procedures and variables are buried in "RE.ST. There is one possibility we haven't covered: you may have some unburied packages of your own in the workspace. If they exist, they will join :LIST in being copied onto disk when the SAVE command is given. Well, we just can't cover every possibility! Just remember to bury or erase your packages before using STORE.DATA.

Other points to notice are that we give the user the option of abandoning the project and that we tidily erase :LIST once we are done with it.

In Fig. 13-10, we gather all the parts of the program together, aside from the utilities from earlier chapters. Fig. 13-11 clarifies the organization with a box chart.

```
TO STORE.DATA
(LOCAL "MENU "FILE.NO "FILENAME
INIT
GET.DATA
SAVE.DATA
END

   TO INIT
   CLEARTEXT RECYCLE
   PR [STORE.DATA CREATES OR ADDS TO FILES]
   PR [THAT WILL BE STORED ON DISK. IT CAN]
   PR [MAKE 2 DIFFERENT TYPES OF FILES:]
   PR []
   PR [1) A LIST OF NUMBER PAIRS, EG, X-Y]
   PR [VALUES FOR GRAPH PLOTTING.]
   PR []
   PR [2) A LIST OF ELEMENTS, EG, NUMBERS]
   PR [FOR A HISTOGRAM, OR WORDS FOR A LIST.]
   PR []
   PR [WHICH TYPE OF FILE DO YOU WANT? 1 OR 2?]
   MAKE "FILE.NO GETB.12
   END

   TO GET.DATA
   CLEARTEXT
   PR [WHICH DO YOU WANT TO DO?]
   PR []
   PR [1) CREATE A NEW FILE]
   PR [2) ADD TO AN EXISTING FILE]
   PR []
   PR [PLEASE ENTER A 1 OR A 2]
   MAKE "MENU GETB.12
   GET.NAME
   IF AND :FILE.NO = 1 :MENU = 1 [MAKE "LIST []
      ADD.PTS]
   IF AND :FILE.NO = 1 :MENU = 2 [ADD.PTS]
   IF AND :FILE.NO = 2 :MENU = 1 [PR [TYPE IN YOUR
      LIST OF DATA] MAKE "LIST GETC.LIST]
   IF AND :FILE.NO = 2 :MENU = 2 [ADD.LIST]
   END
```

Fig. 13-10. STORE.DATA program.

```
TO GET.NAME
PR []
PR [WHAT IS THE FILE NAME?]
MAKE "FILENAME GETB.WORD
PR []
IF :MENU = 1 [STOP]
LOAD :FILENAME
PR []
PR [HERE ARE THE EXISTING DATA:]
PR :LIST
END

TO ADD.PTS
LOCAL "ENTRY
PR []
PR [TYPE IN ONE (X,Y) DATA PAIR TO]
PR [BE ADDED.]
PR []
PR [TO QUIT ADDING, HIT A <RETURN>]
PR [WITH NO DATA.]
PR []
MAKE "ENTRY GET.POINT
IF :ENTRY = [] [STOP]
MAKE "LIST LPUT :ENTRY :LIST
PR :LIST
ADD.PTS
END

TO GETC.LIST
LOCAL "ANS
MAKE "ANS RL
IF :ANS = [] [PR [PLEASE ENTER A LIST, THEN <RETURN>]
   OP GETC.LIST]
PR []
PR [IS THIS RIGHT? Y OR N?]
IF GETB.YN = "Y [OP :ANS] [PR [] PR [ENTER
   NEW LIST] OP GETC.LIST]
END

TO ADD.LIST
LOCAL "ENTRY
PR [TYPE IN ALL DATA ELEMENTS TO BE]
PR [ADDED TO THE EXISTING LIST.]
```

Fig. 13-10.–cont. STORE.DATA program.

```
    PR []
    MAKE "ENTRY GETC.LIST
    MAKE "LIST SE :LIST :ENTRY
    PR [HERE IS THE COMPLETE LIST:]
    PR []
    PR :LIST
    WAIT 300
    END

TO SAVE.DATA
LOCAL "ANS
PACKAGE "STORE.DATA [STORE.DATA SAVE.DATA GET.NAME
    GET.DATA INIT ADD.PTS ADD.LIST GETB.YN GETC.LIST
    GETB.12 GETB.WORD]
BURY "STORE.DATA
PACKAGE "LI.ST [LIST]
PKGALL "RE.ST
BURY "RE.ST
CLEARTEXT
PR [ALL PROCEDURES ARE NOW TEMPORARILY BURIED]
PR []
(PR [I'M READY TO SAVE DATA IN] :FILENAME)
PR [IS THIS WHAT YOU WANT? Y OR N?]
MAKE "ANS GETB.YN
IF :ANS = "N [PR [OK, MAYBE SOME OTHER TIME.]
    STOP]
PR [HERE GOES!]
IF :MENU = 2 [PR [GOT TO ERASE THE OLD FILE
    FIRST] ERASEFILE :FILENAME
SAVE :FILENAME
ERN "LIST
PR [DONE!]
UNBURY "STORE.DATA
UNBURY "RE.ST
PR [ALL PROCEDURES ARE RESTORED]
END
```

Fig. 13-10.–cont. STORE.DATA program.

13.12 An Example

Let's see STORE.DATA at work.

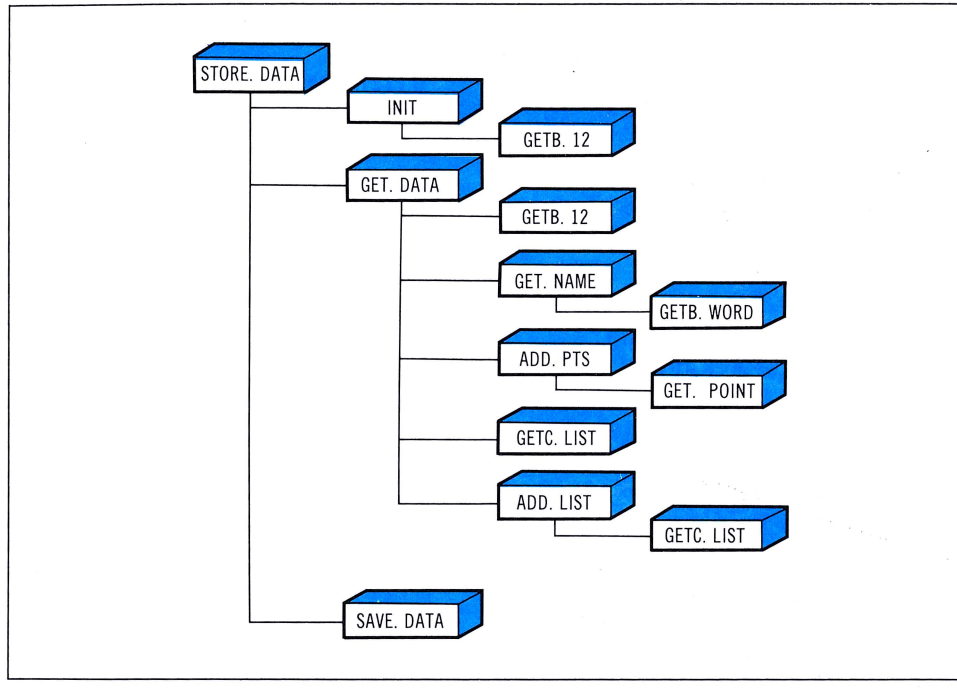


Fig. 13-11. The STORE.DATA box chart.

Example

```
?STORE.DATA
STORE.DATA CREATES OR ADDS TO FILES
THAT WILL BE STORED ON DISK. IT CAN
MAKE 2 DIFFERENT TYPES OF FILES:

1) A LIST OF NUMBER PAIRS, EG, X-Y
VALUES FOR GRAPH PLOTTING.

2) A LIST OF ELEMENTS, EG, NUMBERS
FOR A HISTOGRAM, OR WORDS FOR A LIST.

WHICH TYPE OF FILE DO YOU WANT? 1 OR 2?
2
WHICH DO YOU WANT TO DO?

1) CREATE A NEW FILE
2) ADD TO AN EXISTING FILE
```

```
PLEASE ENTER A 1 OR A 2
1
WHAT IS THE FILE NAME?
ANIMALS
TYPE IN YOUR LIST OF DATA
AARDVARK BEAR COUGAR DONKEY ELAND FERRET GNU HYENA IBEX
IS THIS RIGHT? Y OR N?
Y
ALL PROCEDURES ARE NOW TEMPORARILY BURIED

I'M READY TO SAVE DATA UNDER ANIMALS
IS THIS WHAT YOU WANT? Y OR N?
Y
HERE GOES!
DONE!
ALL PROCEDURES ARE RESTORED
```

At this point, POTS and PONS will show nothing, and CATALOG will reveal a new disk file called "ANIMALS. If you type

```
LOAD "ANIMALS
PONS
```

you will find that :LIST is back and that its contents are the names we typed in.

13.13 Summary

In this chapter we developed two programming projects. These projects provide useful services (graphing and data saving), and they also illustrate what goes into building a program. The key point is modularity. In each case a relatively large program was broken down into smaller, easily managed procedures. In the next chapter we will emphasize this point even more strongly. Meanwhile, why not take the time to play around with these programs. After all, playing is the purest form of learning.

13.14 Review Questions and Exercises

Review Questions

1. Why does GETB.12 check for the empty list ([ ]) before checking to see if the response is a 1 or a 2?

2. How would you increase the graph pointsize in MAKEGRAPH?
3. How would you place the procedures FLIM and FLAM and the list "TARGETS into the package "PK1?

Exercises at the Terminal

1. If you have a color monitor, add a procedure that lets you choose the background and foreground colors.
2. Produce a graph of your local annual rainfall for the last 10 years.
3. The NUMB? procedure we developed checks to see if the first and last members of a list are numbers. See if you can design a procedure to check if all members of a list are numbers.

Answers to Review Questions

(1) The test for a 1 or 2 uses the expression FIRST RL to convert the response from a list to a number; the FIRST primitive balks at the empty list. (2) In MARK.XY change each FD 1 to, say, FD 3 and each BK 2 to, say, BK 6. (3) PACKAGE "PK1 [FLIM FLAM TARGETS]

CHAPTER 14

HANGMAN

CONCEPTS:

planning a program
integrating graphics and text

14.0 Introduction

Let's play! Playing refreshes the spirit and enlivens the mind. Sometimes it soothes, some times it excites, some times it teaches. If one couldn't play with computers, computers would not have progressed as rapidly as they have. Sure, it's great to use computers to store data and to solve problems, but don't forget to play with your computer.

One way to play, of course, is use a computer game. In this chapter we will show how to recreate an old and popular game on the computer. The game is Hangman, although perhaps these days we should refer to it as Hangperson or Hangentity. In addition to giving us a game we can play, this project will give us an opportunity to practice our program-development skills. We will encounter some common programming problems and find solutions for them.

First, let's review how the game works. One player chooses a word, sketches a gallows, and then indicates the number of letters in the word by drawing an equal number of dashes. The second player then begins guessing letters. If the letter occurs in the word, the first player places the letter wherever it occurs in the word. If the letter does not occur in the word, the first player draws in part of a body suspended from the gallows. First comes the head, then the neck, and so on. The second player wins if he gets the complete word before the gallows gets the complete body.

The game is fairly simple, but it requires the computer to do a variety of tasks. For example, the computer must choose a word, decide if a letter belongs, and do some drawing. Clearly, a Logo program to do all this will involve many procedures. Therefore we should do some organizational work before doing the actual programming.

14.1 A Hangman Overview

Let's begin by outlining what the program should do.

1. Explain itself.
2. Choose a word.
3. Draw the gallows.
4. Draw in dashes to represent the word.
5. Get the player's guess and process it.
6. Continue with step 5 until the player wins or loses.
7. Offer the player another game.

Some of these steps correspond to single procedures, but steps 5 and 6 will require further subdivision. We'll deal with that when we get there. For now, we can start with the following Logo representation of a Hangman program

Listing

```
TO HANGMAN1
BEGIN
CHOOSEWORD
SETSCENE
DASHMAKE
GETGUESS
PROCESS
IF NOT CONTINUE? [PR [BYE!] CS CLEARTEXT STOP]
HANGMAN1
END
```

Here we have used the CONTINUE? procedure we developed in chapter 12. Fig. 14-1 presents a box chart of our initial Hangman plan.

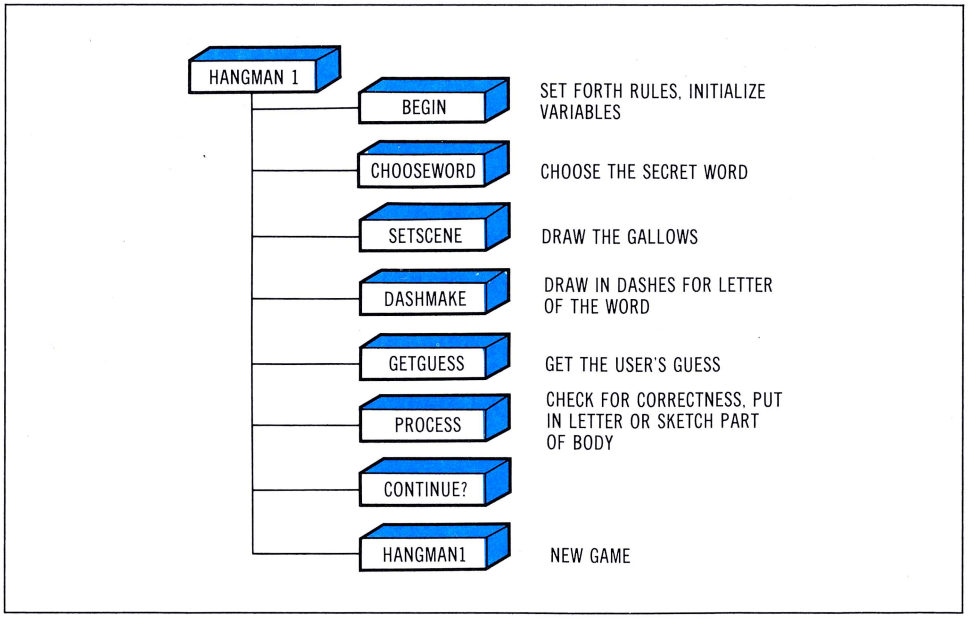


Fig. 14-1. Planning box chart for Hangman.

Now we have the beginning of the structure for the program, although we may have to make some changes as we get into the details. Next, we should start thinking about what variables we will need. Clearly, we need a list of words to choose from. Then we need a variable to represent the chosen word. To decide whether or not a guess is correct, it would be useful to have a list of the letters in the secret word. To tell whether the player wins or loses, we should keep track of the number of right guesses and of the number of wrong guesses. To be fair, we should keep track of past guesses to help screen out repeat guesses. Finally, it would help the player to display a list of his wrong guesses. Okay, let's choose names.

Variables

`"WORDS`	List of words
`"CHOICE`	Chosen word from "WORD list
`"RIGHT`	Number of right guesses
`"WRONG`	Number of wrong guesses
`"GUESSLIST`	List of past guesses
`"RIGHTLIST`	List of letters in "CHOICE
`"WRONGLIST`	List of player's wrong guesses

Notice that several of these variables need to be initialized at the beginning of the program and then be updated as play progresses. One way to handle this is to declare all these variables as LOCAL in the main Hangman procedure. Then they will be private to the program and still be available to all the subprocedures. Thus our first revision to the main program is to make it look like this:

Listing

```
TO HANGMAN2
(LOCAL "WORDS "CHOICE "RIGHT "WRONG "GUESSLIST "RIGHTLIST
   "WRONGLIST)
BEGIN
CHOOSEWORD
SETSCENE
DASHMAKE
GETGUESS
PROCESS
IF NOT CONTINUE? [PR [BYE!] CS CLEARTEXT STOP]
HANGMAN2
END
```

Note: You may wish to omit the LOCAL statement when developing the program. That makes it easier to check on the values of these variables; for if they are not local, they will continue to exist after you leave the program.

Let's turn to the individual procedures.

14.2 The BEGIN Procedure

First, let's outline what we wish to accomplish.

1. Clear the screen and print a heading at the top.
2. Print out the instructions.
3. Initialize variables.

None of this is too difficult, and here is our procedure. Note that we use some of the formatting utilities we developed in Chapter 12.

Listing

```
TO BEGIN
```

```
TEXTSCREEN
CLEARTEXT
SC 0 0
SP 16
PR "HANGMAN
PR [ ]
WAIT 100
PR [I'LL CHOOSE A WORD, AND YOU TRY TO]
PR [GUESS IT BY CHOOSING LETTERS.]
PR [ ]
PR [7 WRONG GUESSES AND YOU LOSE.]
MAKE "WORDS [APPLE BARGE CANOE DEMAND EFFORT GAUGE HOUSE
   INEPT JACKEL KNIFE LOGO MANGO NICELY OPERA PIANO
   QUART RAPID SILLY]
MAKE "WRONG 0
MAKE "RIGHT 0
MAKE "WRONGLIST [ ]
MAKE "RIGHTLIST [ ]
MAKE "GUESSLIST [ ]
WAIT 300
END
```

We use the Logo primitive WAIT to give the user time to read the messages. The command WAIT 60 causes the program to wait one second before going on to the next step. WAIT 120 waits two seconds, and so on. You can check this procedure by running it. It should print out the information as given. You can check to see if the initializations worked by following up with commands such as

```
PR :WRONG
```

or

```
PR FIRST :WORDS
```

You can, of course, make the word list longer or shorter, harder or easier.

The next step is to choose a word from the list.

14.3 The CHOOSEWORD Procedure

First, we outline what the procedure should do.

1. Choose a word from the word list and assign it to the variable :CHOICE.
2. Prepare the list of letters in the chosen word; that is, identify the letters in the word and make a list of them.

Here is one approach:

Listings

```
TO CHOOSEWORD
MAKE "CHOICE RANPICK :WORDS
GOODLIST :CHOICE
END

TO RANPICK :LIST
OUTPUT ITEM (1 + RANDOM (COUNT :LIST)) :LIST
END

TO GOODLIST :WORD
IF :WORD = " [STOP]
TEST MEMBERP FIRST :WORD :RIGHTLIST
IFFALSE [MAKE "RIGHTLIST LPUT FIRST :WORD :RIGHTLIST]
GOODLIST BUTFIRST :WORD
END
```

We've set up RANPICK as a general purpose utility. Give it a list as input, and it selects a member at random and outputs it. Let's review how it works.

First, the Logo primitive COUNT counts how many items the input list contains; call it n. Then the Logo primitive picks a random number in the range 0 to n − 1. We add 1 to provide a number in the range 1 to n. Then the Logo primitive ITEM extracts that item from the list. For example, if 5 is the number finally selected, the 5th item is plucked from the list.

Our GOODLIST procedure sets up the list of correct letters. It checks to see if the first letter of the chosen word is in :RIGHTLIST. If not, it uses the Logo primitive LPUT to place that letter in the list. Then, in a recursive call, it strips off the first letter from the word and repeats the process. (Now the second letter becomes the new first letter.) This continues until nothing is left but the empty word ". Notice that duplicate letters get included just once. If, for example, :CHOICE is "BOOKKEEPER, then :RIGHTLIST is [B O K E P R].

Now let's draw the gallows.

14.4 The SETSCENE Procedure

We require that this procedure empty the screen and draw a simple gallows.

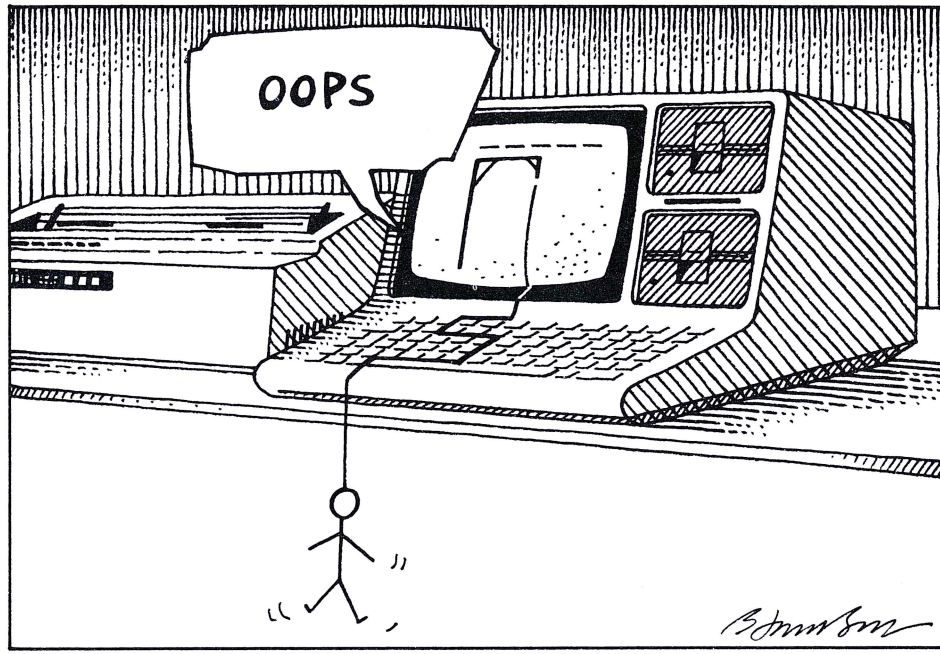
Listing

```
TO SETSCENE
CLEARTEXT
CS
HT
PU
SETHEADING 0
SETPOS [-100 65]
PD
FR 30
FR 90
FR 30
FR 90
FR 180
FR 50
FD 20
END

TO FR :X
FD :X
RT 90
END
```

The drawing consists of FORWARD moves combined with right-angle turns, so we created the auxiliary procedure FR to handle that task. Fig. 14-2 shows the resulting picture.

Fig. 14-2. The gallows.

Now that we have the drawing, let's put in the dashes for the mystery word.



14.5 The DASHMAKE Procedure

This procedure needs to count the number of letters in the chosen word and to then place that number of dashes below the gallows. Because :CHOICE is declared LOCAL in the main procedure, it will be known to this procedure without naming it as an input. However, we can name it as an input if we like. Then we will have a procedure we could use with variables other than just :CHOICE. Perhaps we will never need such a procedure again, but it usually is a good idea to make procedures as modular as possible. Thus, in our Hangman program, we now replace the line

```
DASHMAKE
```

with

```
DASHMAKE :CHOICE
```

Here is the procedure along with its supporting cast:

Listing

```
TO DASHMAKE :WORD
LOCAL "LEN
LOCAL "X
MAKE "LEN COUNT.W :WORD
MAKE "X 5
DASHOUT :X :LEN
END

TO COUNT.W :WORD
IF EMPTYP :WORD [OUTPUT 0]
OUTPUT 1 + COUNT.W BF :WORD
END

TO DASHOUT :X :LEN
IF :LEN = 0 [STOP]
SC :X 20
TYPE "-
DASHOUT :X + 2 :LEN -1
END
```

Here is a sample run showing the gallows and the dashes produced for the mystery word "canoe."

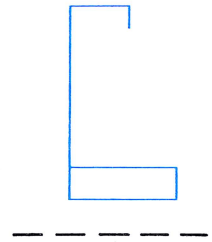


Fig. 14-3. Dashes and gallows.

DASHMAKE establishes two local variables :LEN and :X to keep track of the length of the word and of the position where a dash is to be placed. The utility COUNT.W is used to count the number of letters in our word. (We discussed this utility in an earlier chapter.) Then DASHOUT does the actual printing of the dashes.

DASHOUT takes two inputs. The first is the column position of the dash. The second is the number of dashes to be printed. The procedure calls itself recursively, shifting the position two columns to the right each time and subtracting one from the number of dashes to be printed.

Note that the dashes are placed on the 21st line (the lines are numbered from 0 to 23). We will have to return to that line when we print out the correct guesses.

Now we get to the heart of the program, getting and processing the guesses.

14.6 The GETGUESS and PROCESS Procedures

Before we attack the details of these procedures, we should note that this part of the program will have to be cycled through several times until we have a winner or loser. That suggests that these two procedures be embedded in a single procedure that calls itself recursively until the number of wrong guesses is 7 or the number of right guesses equals the number of letters in :RIGHTLIST. Thus, in our tentative HANGMAN program, we replace the lines

```
GETGUESS
PROCESS
```

with

```
GUESSPART
```

Here is the listing for GUESSPART.

Listing

```
TO GUESSPART
LOCAL "ANS
MAKE "ANS GETGUESS
PROCESS :ANS
IF OR (:WRONG = 7) (:RIGHT = COUNT :RIGHTLIST) [STOP]
GUESSPART
END
```

Now we need to attend to the details. Let us begin with GETGUESS. Here is a first pass:

Listing

```
TO GETGUESS1
LOCAL "ANS
SC 3 22
TYPE [ENTER YOUR GUESS, PLEASE.]
MAKE "ANS READCHAR
OUTPUT :ANS
END
```

Notice that we use READCHAR to get the user's answer. This means she doesn't have to type RETURN after typing her choice.

This procedure is passable, but let's fix it so that it will screen out previous guesses. We can do this by comparing the guess to the :GUESSLIST list, which was initialized to [ ] in BEGIN. If the guess is in the list, let the user know and run him through GETGUESS again. We will update :GUESSLIST in PROCESS, once a new guess is obtained. Here is the revised version:

Listing

```
TO GETGUESS
LOCAL "ANS
SC 3 22
TYPE [ENTER YOUR GUESS, PLEASE.]
MAKE "ANS READCHAR
IF MEMBERP :ANS :GUESSLIST [CLEARLINE 22 PR [YOU'VE USED
   THAT ONE. TRY AGAIN.] WAIT 100 CLEARLINE 22
   OUTPUT GETGUESS]
OUTPUT :ANS
END

TO CLEARLINE :Y
SC 0 :Y
REPEAT 39 [TYPE CHAR 32]
SC 0 :Y
END
```

We use CLEARLINE to keep our new message from overlapping the old.

This is not the last word for GETGUESS. If you like, you could also screen out replies such as 4, $, or RETURN. But we leave that to you. Let's go on to PROCESS.

Listing

```
TO PROCESS :ANS
MAKE "GUESSLIST LPUT :ANS :GUESSLIST
TEST MEMBERP :ANS :RIGHTLIST
IFTRUE [MAKE "RIGHT :RIGHT + 1 FINE :CHOICE :ANS]
IFFALSE [MAKE "WRONG :WRONG +1 OOPS :ANS]
END
```

This is straightforward. First, as promised, we update the list of guesses. Then we check to see if the guess matches our list of correct letters. If so, we up the :RIGHT count by 1 and turn control over to the FINE procedure. If not, we up the :WRONG count by 1 and go to OOPS.

Now we are getting procedures within procedures within procedures. To help clarify matters, here is a box chart of what we've done so far. Next, we'll look at FINE and OOPS.

14.7 The FINE Procedure

The FINE procedure is activated if the player makes a correct guess. We assign it these tasks:

1. Find which position or positions the correct letter occupies in the word.
2. Replace the corresponding dashes with the correct letter.
3. Congratulate the player if he gets the word.

Again we will invoke subprocedures.

Listing

```
TO FINE :CHOICE :ANS
LOCAL "PLACE
MAKE "PLACE [ ]
LOCATE :CHOICE :ANS 1
```

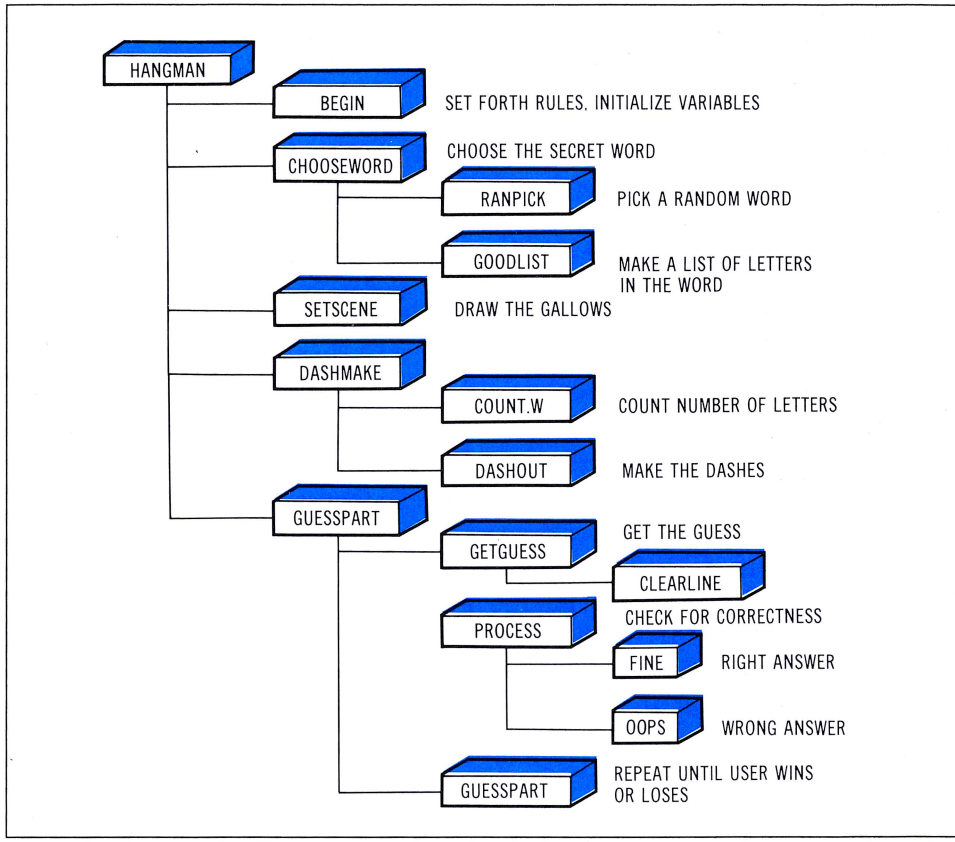


Fig. 14-4. Partial box chart.

```
PUTANS :ANS :PLACE
IF :RIGHT = COUNT :RIGHTLIST [SC 20 21 PR [YOU GOT IT!]]
END

TO LOCATE :WORD :LETTER :N
IF :WORD = '' [STOP]
IF :LETTER = FIRST :WORD [MAKE "PLACE LPUT :N :PLACE]
LOCATE BF :WORD :LETTER :N + 1
END

TO PUTANS :LETTER :LIST
LOCAL "XP
IF :LIST = [ ] [STOP]
MAKE "XP ((FIRST :LIST) * 2) + 3
```

```
SC :XP 20
TYPE :LETTER
PUTANS :LETTER BF :LIST
END
```

What's happening here? First, FINE defines a local variable :PLACE and sets it equal to the empty list. Then LOCATE, using recursion, looks through the mystery word a letter at a time, seeing if the letter matches the guess. Whenever there is a match, the numerical position of the letter is added to the :PLACE list. For example, if :CHOICE is "silly" and the player guesses "l," then :PLACE will equal [3 4], since "1" occupies the 3rd and 4th places in the word.

Next, the PUTANS procedure puts the player's answer in the correct spaces. PUTANS uses recursion to go through the :PLACE list one number at a time until it exhausts the list. If it finds, say, the number 3, then it moves the cursor to column 3*2+3; that is, column 9, and replaces the dash there with the player's guess. Remember that DASHMAKE put dashes in columns 5, 7, 9, etc. The *2 in the formula moves the cursor over 2 at a time, and the +3 makes the first letter fall in column 5.

Finally, FINE tells the player that she won once she has guessed all the letters in :RIGHTLIST.

And what if the player is wrong? Then the program takes the OOPS option.

14.8 The OOPS Procedure

This procedure has the following duties:

1. Update the list of wrong answers.
2. Inform the player that he is wrong.
3. Draw the next installment of the suspended body.
4. Display the list of wrong guesses.
5. Make appropriate warnings and comments.

Here it goes:

Listing

```
TO OOPS :ANS
```

```
MAKE "WRONGLIST LPUT :ANS :WRONGLIST
SC 3 21
PR SE :ANS [IS WRONG]
DRAWFIG :WRONG
SC 3 23
(TYPE [WRONG LIST:] CHAR 32 :WRONGLIST)
IF :WRONG = 6 [CLEARLINE 22 PR [BE CAREFUL!] WAIT 100
   CLEARLINE 22]
IF :WRONG = 7 [FINISHUP]
END

TO DRAWFIG :N
LOCAL "BODYPARTS
MAKE "BODYPARTS [HEAD NECK ARM1 ARM2 BODY LEG1 LEG2]
RUN LIST (ITEM :N :BODYPARTS)
END

TO FINISHUP
REPEAT 4 [TYPE CHAR 7]
CLEARLINE 21
CLEARLINE 22
CLEARLINE 23
SC 3 21
(PR [THE WORD WAS] :CHOICE)
WAIT 100
END
```

The one aspect that requires some explanation (and some more procedures) is the figure drawing. The procedure DRAWFIG takes as input the number of wrong answers; call it n. The list :BODYPARTS contains the names of seven procedures for drawing the seven consecutive body parts to be added to the figure. The ITEM primitive then selects the name of the nth procedure, and the RUN primitive then runs it. For example, the fourth wrong answer will cause the fourth procedure (ARM2) to be run.

Here are the procedures for drawing the body.

Listing

```
TO HEAD
LT 90
REPEAT 54 [FD 3 RT 10]
LT 90
END
```

```
TO NECK
FD 20
END

TO ARM1
RT 120
FD 30
BK 30
RT 120
END

TO ARM2
FD 30
BK 30
RT 120
END

TO BODY
FD 40
END

TO LEG1
LOCAL "POS1
MAKE "POS1 POS
RT 45
FD 25
RT 90
FD 10
PU
SETPOS :POS1
SETH 180
PD
END

TO LEG2
LT 65
FD 25
LT 90
FD 10
END
```

Fig. 14-5 shows the finished product. Feel free to adjust the figure to suit your own aesthetic tastes.

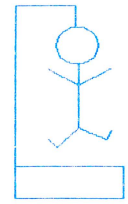


Fig. 14-5. The finished gamester.

This about winds the program up. We just have to ask the user if he wants to play again.

Listing

```
TO CONTINUE?
PR [PLAY ANOTHER GAME? <Y OR N>]
IF GETB.YN = "Y [OP "TRUE] [OP "FALSE]
END
```

14.9 The Hangman Procedure

We made some changes in the program as we went along, so let's present our final version of the program in Fig. 14-6.

```
TO HANGMAN
(LOCAL "WORDS "CHOICE "RIGHT "WRONG "GUESSLIST "RIGHTLIST
   "WRONGLIST)
BEGIN
CHOOSEWORD
SETSCENE
DASHMAKE :CHOICE
GUESSPART
IF CONTINUE? = "FALSE [PR [BYE!] CS CLEARTEXT STOP]
HANGMAN
END

   TO BEGIN
   TEXTSCREEN
```

Fig. 14-6. The Hangman program.

```
CLEARTEXT
SC 0 0
SP 16
PR "HANGMAN
PR [ ]
WAIT 100
PR [I'LL CHOOSE A WORD, AND YOU TRY TO]
PR [GUESS IT BY CHOOSING LETTERS.]
PR [ ]
PR [7 WRONG GUESSES AND YOU LOSE.]
MAKE "WORDS [APPLE BARGE CANOE DEMAND EFFORT GAUGE HOUSE
      INEPT JACKEL KNIFE LOGO MANGO NICELY OPERA PIANO
      QUART RAPID SILLY]
MAKE "WRONG 0
MAKE "RIGHT 0
MAKE "WRONGLIST [ ]
MAKE "RIGHTLIST [ ]
MAKE "GUESSLIST [ ]
WAIT 300
END


TO CHOOSEWORD
MAKE "CHOICE RANPICK :WORDS
GOODLIST :CHOICE
END


   TO RANPICK :LIST
   OUTPUT ITEM (1 + RANDOM (COUNT :LIST)) :LIST
   END


   TO GOODLIST :WORD
   IF :WORD = " [STOP]
   TEST MEMBERP FIRST :WORD :RIGHTLIST
   IFFALSE [MAKE "RIGHTLIST LPUT FIRST :WORD :RIGHTLIST]
   GOODLIST BUTFIRST :WORD
   END
```

Fig. 14-6.–cont. The Hangman program.

```
TO SETSCENE
CLEARTEXT
CS
HT
PU
SETPOS [0 -60]
PD
FR 30
FR 90
FR 30
FR 90
FR 180
FR 50
FD 20
END


   TO FR :X
   FD :X
   RT 90
   END


TO DASHMAKE :WORD
LOCAL "LEN
LOCAL "X
MAKE "LEN COUNT.W :WORD
MAKE "X 5
DASHOUT :X :LEN
END


   TO COUNT.W :WORD
   IF EMPTYP :WORD [OUTPUT 0]
   OUTPUT 1 + COUNT.W BF :WORD
   END


   TO DASHOUT :X :LEN
   IF :LEN = 0 [STOP]
   SC :X 20
```

Fig. 14-6.–cont. The Hangman program.

```
   TYPE "-
   DASHOUT :X + 2 :LEN - 1
   END


TO GUESSPART
LOCAL "ANS
MAKE "ANS GETGUESS
PROCESS :ANS
IF OR (:WRONG = 7) (:RIGHT = COUNT :RIGHTLIST) [STOP]
GUESSPART
END


TO GETGUESS
LOCAL "ANS
SC 3 22
TYPE [ENTER YOUR GUESS, PLEASE.]
MAKE "ANS READCHAR
IF MEMBERP :ANS :GUESSLIST [CLEARLINE 22 PR [YOU'VE USED
THAT ONE. TRY AGAIN.] WAIT 100 CLEARLINE 22 OUTPUT
GETGUESS]
OUTPUT :ANS
END


   TO CLEARLINE :Y
   SC 0 :Y
   REPEAT 39 [TYPE CHAR 32]
   SC 0 :Y
   END


TO PROCESS :ANS
MAKE "GUESSLIST LPUT :ANS :GUESSLIST
TEST MEMBERP :ANS :RIGHTLIST
IFTRUE [MAKE "RIGHT :RIGHT + 1 FINE :CHOICE :ANS]
IFFALSE [MAKE "WRONG :WRONG +1 OOPS :ANS]
END


   TO FINE :CHOICE :ANS
   LOCAL "PLACE
   MAKE "PLACE [ ]
```

Fig. 14-6.–cont. The Hangman program.

```
   LOCATE :CHOICE :ANS 1
   PUTANS :ANS :PLACE
   IF :RIGHT = COUNT :RIGHTLIST [SC 20 21 PR [YOU GOT
      IT!]]
   END


      TO LOCATE :WORD :LETTER :N
      IF :WORD = '' [STOP]
      IF :LETTER = FIRST :WORD [MAKE "PLACE LPUT :N
         :PLACE]
      LOCATE BF :WORD :LETTER :N + 1
      END


      TO PUTANS :LETTER :LIST
      LOCAL "XP
      IF :LIST = [ ] [STOP]
      MAKE "XP ((FIRST :LIST) * 2) + 3
      SC :XP 20
      TYPE :LETTER
      PUTANS :LETTER BF :LIST
      END


TO OOPS :ANS
MAKE "WRONGLIST LPUT :ANS :WRONGLIST
SC 3 21
PR SE :ANS [IS WRONG]
DRAWFIG :WRONG
SC 3 23
(TYPE [WRONG LIST:] CHAR 32 :WRONGLIST)
IF :WRONG = 6 [CLEARLINE 22 PR [BE CAREFUL!] WAIT 100
CLEARLINE  22]
IF :WRONG = 7 [FINISHUP]
END


   TO DRAWFIG :N
   LOCAL "BODYPARTS
   MAKE "BODYPARTS [HEAD NECK ARM1 ARM2 BODY LEG1 LEG2]
   RUN LIST (ITEM :N :BODYPARTS)
   END
```

Fig. 14-6.–cont. The Hangman program.

```
TO FINISHUP
REPEAT 4 [TYPE CHAR 7]
CLEARLINE 21
CLEARLINE 22
CLEARLINE 23
SC 3 21
(PR [THE WORD WAS] :CHOICE)
WAIT 100
END


   TO HEAD
   LT 90
   REPEAT 54 [FD 3 RT 10]
   LT 90
   END


   TO NECK
   FD 20
   END


   TO ARM1
   RT 120
   FD 30
   BK 30
   RT 120
   END


   TO ARM2
   FD 30
   BK 30
   RT 120
   END


   TO BODY
   FD 40
   END
```

Fig. 14-6.–cont. The Hangman program.

```
TO LEG1
LOCAL "POS1
MAKE "POS1 POS
RT 45
FD 25
RT 90
FD 10
PU
SETPOS :POS1
SETH 180
PD
END


TO LEG2
LT 65
FD 25
LT 90
FD 10
END
```

Utilities called:

SP :X
SC :X :Y
GETB.YN

Fig. 14-6.–cont. The Hangman program.

14.10 Closing Comments

We now have a working version of Hangman. By no means is it the last word (see the Exercises for some suggested changes), but it gets the job done. In the process, we have used some useful general utilities, such as RANPICK and COUNT.W. In the design of DRAWFIG we've seen how to have a procedure choose which of several other procedures are run. (Incidentally, that approach can be combined with RANPICK to produce a procedure that picks other procedures at random to run!) Most importantly, we've seen that the way to solve a large problem is to break it down into smaller, more manageable problems.

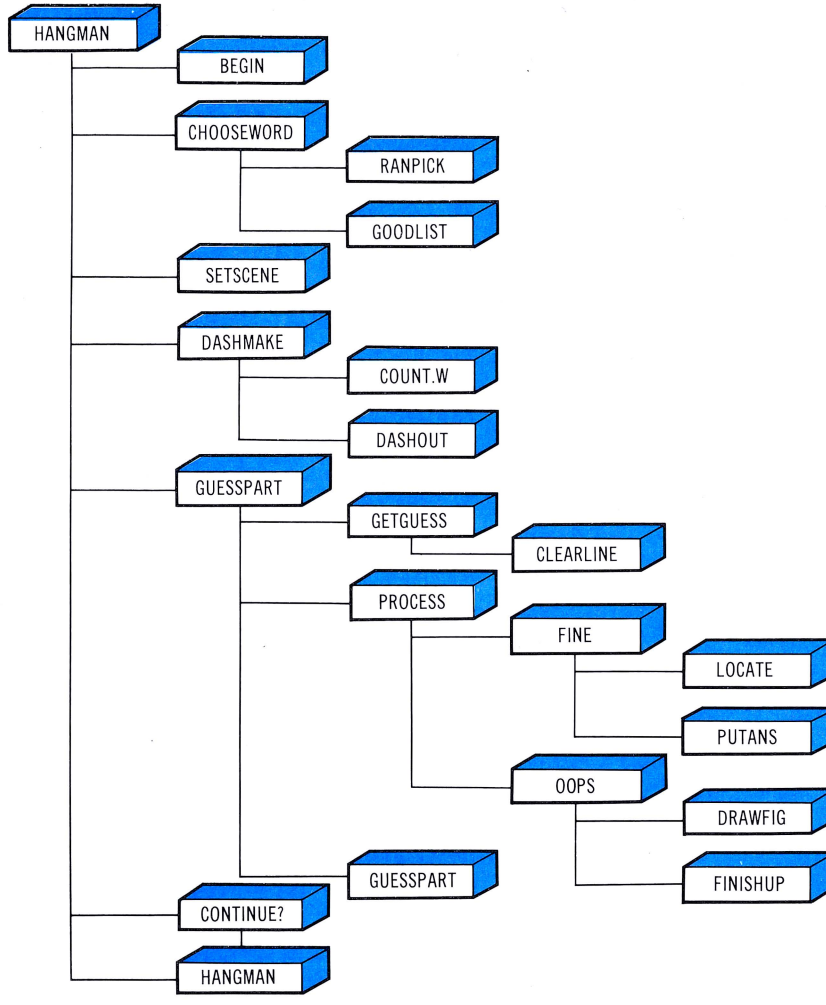


Fig. 14-7. Hangman box chart.

14.11 Review Questions and Exercises

Review Questions

1. The variables "RIGHT and "WRONG are initialized to 0 in BEGIN. Could these variables be declared LOCAL to BEGIN?
2. What is the difference between "CHOICE and "RIGHTLIST?

Exercises at the Terminal

1. Modify GETGUESS so that it screens out responses that are not letters. (Hint: The ASCII primitive gives the ASCII code of a character. Nonletters have an ASCII code less than 65 or greater than 95.)
2. If you have a color monitor, put color into the graphics portion: one color for the background, another color for the gallows, and another color for the victim.
3. Have the computer beep after each response.
4. In this version, GETGUESS uses READCHAR, which doesn't require the use of the RETURN key; the CONTINUE? uses the utility GETB.YN, which does require use of the RETURN key. Decide which mode you prefer, and change the program so that all responses are in the same mode.
5. Modify the program so that it keeps track of wins and losses.
6. The list of mystery words can only be about three lines long. You can create three and four lists of words, save each list separately to disk and have the computer randomly select one and load it. Each list should be called "WORD but the versions on disk could be called WORD1, WORD2, WORD3, etc.

Answers to Review Questions

(1) No, because PROCESS needs to know these values also. (2) The value :CHOICE is a word, the secret word. The value :RIGHTLIST is a list containing the letters of the word; letters appearing more than once in the word appear just once in the list.

CHAPTER 15

THE STATES AND CAPITALS GAME: PROPERTY LISTS

CONCEPTS:

property lists
data bases

PRIMITIVES:

PPROP, GPROP, PLIST, PPS, REMPROP

15.0 Introduction

Here is the 64 dollar question:

"What is the single most important application of computers in the world today?"

The 128 dollar answer:

"Data bases"

Far and away, the most common use of computers is to store information. Think of all the organizations that use quantities of data: banks, schools, government, the IRS (ugh), the mail order catalog industry, supermarkets, politicians, professional sports—the world is full of information just looking for a home.

The data base provides it.

A data base is a warehouse of information. We have seen how Logo stores data in the form of numbers, words, and lists. Now, we have a new way to store information: the property list. we will begin by using property

lists to store information associated with a game, called the "States and Capitals" game.

The purpose of the States and Capitals game is to help one learn the capitals of the various states. (This knowledge could be very valuable if you become, say, a lobbyist or a reporter or a quiz show contestant.) It will select a state at random and ask you its capital: "What is the capital of Rhode Island?" If you reply, "PROVIDENCE," you are congratulated, and the program moves on to another state. You can pay $12 to $20 for this game, or you can use Logo to create your own version. Since you already have this book, you may as well do the latter.

How do we go about this task? By now we have most of the basic tools. We know how to use RANDOM to make random selections. We know how to get a user's answer and to use a conditional to see if it is right or not. We know how to keep score. The new problem presented by this game is finding the best way to store the basic information, the names of the states and the capitals.

The most obvious answer (but not the one we will use!) is to use lists. There are several approaches that could work. We could have one list of 50 states and a second list of 50 capitals. With RANDOM and ITEM, we could pick a state from the first list. By keeping track of the value of ITEM, we then could fetch the corresponding capital from the second list. (We have to make sure that capitals are in the same order as the states). But wait, a state like North Dakota or a capital like Baton Rouge would appear as *two* items on a list, since each is two words. We would need to use a list of lists for each set:

```
[ . . . [NORTH DAKOTA] [OHIO] . . . ]
[ . . . [BISMARCK] [COLUMBUS] . . . ]
```

The method grows more cumbersome. Another approach would be to combine the states and capitals into one master list:

```
[ . . . [[NORTH DAKOTA] [BISMARCK]] [[OHIO] [COLUMBUS]]
. . . ]
```

Use RANDOM and PICK to choose a member, such as

```
[[OHIO] [COLUMBUS]]
```

and then use FIRST and LAST to extract the question and the answer. Either method would work, so why won't we be using them? If we have a

large list of data, it is slow to use FIRST and BUTFIRST to go through the list item by item. There is a better way: using property lists.

Property lists let us group variables and their values together in a new kind of list. Once we explore the way they work, we can resume with our States and Capitals project. The first step, though, is to look at the special primitives that work with property lists. This is important to do, because the ordinary list primitives (FIRST, ITEM, etc.) do not work with property lists.

15.1 Property List Primitives

We have five primitives to study: PPROP, GPROP, PLIST, PPS, and REMPROP. PPROP creates property lists, so by studying it, we should get a good idea of what property lists are. PPROP takes three inputs. For example,

```
?PPROP "VERNA "HAIR "BROWN
?PPROP "VERNA "EYES "BROWN
?PPROP "VERNA "WEIGHT 134
?PRROP "VERNA "HEIGHT [5 FT 4 IN]
```

What have we wrought? Using PPS (for print properties) will give a clue:

```
?PPS
VERNA'S HEIGHT IS [5 FT 4 IN]
VERNA'S WEIGHT IS 134
VERNA'S EYES IS BROWN
VERNA'S HAIR IS BROWN
```

My goodness, all the information came back. It's time to say what is happening. The first input to PPROP is the name or title of the property list. In this example, all our entries applied to the same property list, "VERNA. The second input is the name of a property. In our example, we created four different properties: "HAIR, "EYES, "WEIGHT, and "HEIGHT. The third and final input is a value for the property. The first two inputs must be words, but the last input can be any valid data object. For instance, we used words, a number, and a list as inputs.

Earlier we said a property list lets you group variables and values together in a new kind of list. Here, in one property list ("VERNA), we have grouped together four variables (or properties) and their values.

If we need to keep track of these properties (hair color, etc.) for many people, we could set up a procedure to create property lists of that form.

Listing

```
TO DESCRIBE :NAME
PR [HAIR COLOR?]
PPROP :NAME "HAIR RL
PR [EYE COLOR?]
PPROP :NAME "EYES RL
PR [WEIGHT?]
PPROP :NAME "WEIGHT RL
PR [HEIGHT?]
PPROP :NAME "HEIGHT RL
END
```

This is slightly different from our last example, for here all the third inputs are lists. (They are lists because READLIST outputs lists. We could go back to words by using FIRST :RL, but this approach is more flexible. Now, for example, we can use answers like MOUSY BLOND and BLOODSHOT BLUE.) Here is a sample run.

Sample Run

```
?DESCRIBE :MORT
HAIR COLOR?
BLACK
EYE COLOR?
GREEN
WEIGHT?
188
HEIGHT?
6 FT
?
```

Okay, let's use PPS to see what we have.

```
?PPS
VERNA'S HEIGHT IS [5 FT 4 IN]
VERNA'S WEIGHT IS 134
VERNA'S EYES IS BROWN
VERNA'S HAIR IS BROWN
```

```
MORT'S HEIGHT IS [6 FT]
MORT'S WEIGHT IS [188]
MORT'S EYES IS [GREEN]
MORT'S HAIR IS [BLACK]
```

Well, our procedure works, but PPS seems to print *all* property lists. (Actually, PPS can take a package name as an input, and it will then just print the property lists in the package. But it doesn't take the name of a specific property list as an input.)

The primitives GPROP and PLIST give us finer control in fetching information in property lists. PLIST takes a property list name as input and outputs a list of the contents:

```
?SHOW PLIST "VERNA
[HEIGHT [5 FT 4 IN] WEIGHT 134 EYES BROWN HAIR BROWN]
```

We see that the output is a list consisting of a property, then the property's value, the next property, and so on. (SHOW is like PR, but it includes brackets when printing a list.) The output of PLIST is an ordinary list, and we can use the usual list primitives, like FIRST, with it.

GPROP is more specific yet. It takes two inputs: the property list name, and the name of a particular property. It outputs the value of that particular property.

```
?PR GPROP "VERNA "HAIR
BROWN
?PR GPROP "MORT "EYES
GREEN
```

The names of property lists are not ordinary Logo names. For instance, if you try something like

```
PR :VERNA
```

you will be told that :VERNA has no value. If you type ERNS (erase all names), "VERNA and "MORT will be left unscathed. Property list names will, however, be captured by SAVE, and they can be used with PACKAGE.

To get rid of a property list, you must use REMPROP and remove the list property by property. REMPROP takes two inputs: the property list

name and the property name. It then removes that property and its value. For example,

```
?REMPROP "VERNA "EYES
?PR PLIST "VERNA
HEIGHT [5 FT 4 IN] WEIGHT 134 HAIR BROWN
```

The "EYES property and its value of "BROWN are removed from "VERNA. By removing "HEIGHT, "WEIGHT, and "BROWN, we erase all of "VERNA.

To summarize, PPROP creates or adds to a property list. It takes three inputs: the name of the property list, the name of a specific property, and the value of that property. PPS without an input prints out all the property lists in the workspace. It can also take a package name (or list of package names) as input and print out the property lists in them. PLIST takes a property list name as input and outputs a list of all the properties and respective values in the property list. GPROP takes a property list name and a property name as its two inputs and outputs the value of that particular property. Finally, REMPROP takes the same input as GPROP, but removes that property and its value from the property list.

Here is a formal definition of PPROP.

PPROP	**(takes 3 inputs)**
Purpose	Put a property and its value into a property list
Abbrev	None
Inputs	First input is the name of the property list. Second input is the name of a specific property. Third input is the value of the property. The first two inputs must be words, the third can be a word, number, or list
Outputs	None

Examples. (creating a property list with 3 properties)

```
?PPROP "HOUSE "ROOMS 8
?PPROP "HOUSE "COLOR "BLUE
?PPROP "HOUSE "ADDRESS [12906 BRISKET BLVD]
```

Now let's apply what we have learned to the States and Capitals game.

15.2 States and Capitals

Our main interest in this game is using property lists to store the basic information used by the game, so we will begin with that topic. The data we wish to store are the names of states and the names of capitals. As we noted earlier, some of these names are two words, so we will need to use a list for each name. If we use a property list, the only input to PPROP that can be a list is the third input, the property value. This suggests we set up a property list for each state. The first property will be "STATE and will be assigned the name of the state. The second property will be "CAPITAL and will be assigned the name of the capital. Altogether we will have 50 property lists, one for each state and capital pair. But what shall we name the property lists themselves? Since the data form the fodder for 50 possible questions, we will name the property lists "Q1, "Q2, etc.

This technique of using a number in the property list name gives us an easy way to gain access to the data. For instance, the long way of creating the data base would be to do this:

```
PPROP "Q1 "STATE [ALABAMA]
PPROP "Q1 "CAPITAL [MONTGOMERY]
PPROP "Q2 "STATE [ALASKA]
PPROP "Q2 "CAPITAL [JUNEAU]
```

and so on for all 50 states. But we should be able to use the computer to help with repetitive tasks, and we can do so here. Here we create a recursive program that prompts us for the state and capital names and then puts them in the proper property list.

Listing

```
TO MAKE.P.LIST :N
PR [ENTER THE STATE]
PPROP WORD "Q :N "STATE GETC.LIST
PR [ENTER THE CAPITAL]
PPROP WORD "Q :N "CAPITAL GETC.LIST
IF NOT CONTINUE? [STOP]
MAKE.P.LIST :N + 1
END
```

We use the CONTINUE? procedure we used in Chapters 12 and 14.

We start off this way:

Sample Run

```
?MAKE.P.LIST 1
ENTER THE STATE
ALABAMA
ENTER THE CAPITAL
MONTGOMERY
DO YOU WISH TO CONTINUE? Y OR N?
Y
ENTER THE STATE
ALASKA
ENTER THE CAPITAL
JUNEAU
DO YOU WISH TO CONTINUE? Y OR N?
N
?
```

With an input to MAKE.P.LIST of 1, WORD "Q :N becomes "Q1, the name of the first property list. The recursive call of MAKE.P.LIST increases :N to 2, so the next set cycle creates the "Q2 property list. Using WORD lets us count through the whole series of property list names. If we want to continue with the list after stopping (as we did above), just invoke the next number:

```
?MAKE.P.LIST 3
```

and take it from there.

We can think of the numbers in the names as being *subscripts* or *indices* used to number the property lists. The numbers become convenient labels to identify the property lists. We used them to create the lists, but they also can be used to recover information from the lists, as we soon shall see.

Now that we have the data stored in a convenient form, let's use it in a quiz game. We'll call the game STATES. Here is the basic framework:

Listing

```
TO STATES
(LOCAL "RIGHT "GUESSES)
STARTUP
QUIZZEM
SUMUP
END
```

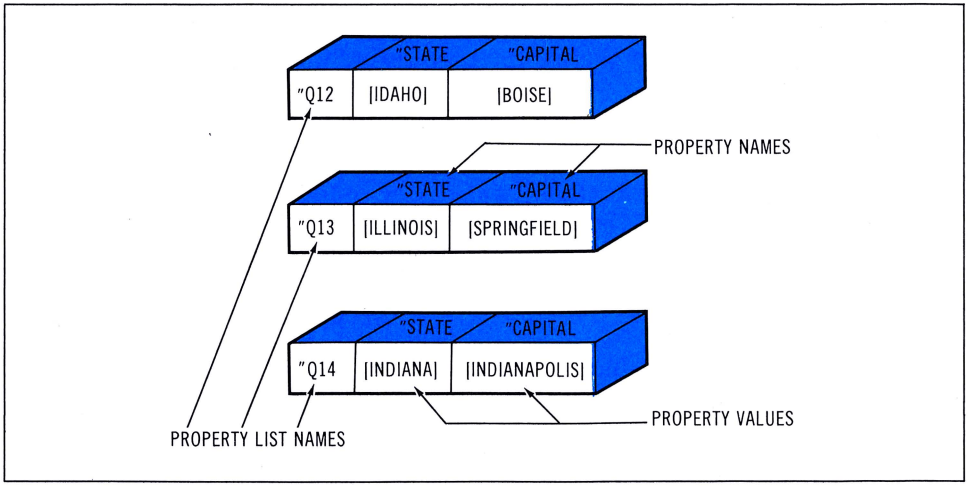


Fig. 15-1. Numbered property lists.

This is pretty simple. STARTUP will start things off, QUIZZEM will handle the questions and answers, and SUMUP will report on the user's success. The local variables :RIGHT and :GUESSES will keep track of how many right answers the user has and of how many guesses he made.

Here is STARTUP:

Listing

```
TO STARTUP
TEXTSCREEN CLEARTEXT SETCURSOR [0 0]
PR [DO YOU KNOW YOUR STATE CAPITALS?]
PR [LET'S FIND OUT. I'LL GIVE YOU THE]
PR [NAME OF A STATE, AND YOU GIVE ME THE]
PR [NAME OF ITS CAPITAL.]
PR [ ]
PR [OK, HERE WE GO!]
MAKE "RIGHT 0
MAKE "WRONG 0
END
```

There is nothing new here. Let's move on to QUIZZEM.

Listing

```
TO QUIZZEM
```

```
(LOCAL "QLIST "ANS "RESPONSE)
MAKE "QLIST WORD "Q (RANDOM 50) + 1
ASK :QLIST
MAKE "ANS GET.CAP :QLIST
MAKE "RESPONSE GETC.LIST
COMPARE :ANS :RESPONSE
IF NOT CONTINUE? [STOP]
QUIZZEM
END
```

First, we produce a random number in the range 1 to 50. Then we use WORD to tag on this number to "Q so we can assign one of the 50 property list names to "QLIST. (We assume you have been diligent and have completed the property lists for all 50 states. Thus the value :QLIST can range from "Q1 to "Q50.) ASK then (see below) fetches the value of the "STATE property from that property list. The local variable :ANS gets assigned the capital from that same list, and the local variable :RESPONSE gets the user's answer. (We use the utility GETC.LIST to give the user a chance to change his mind.) Then COMPARE compares the two and informs the user of his success or failure. (Again, see below.) Then, if the user wants to continue, the whole QUIZZEM process is repeated.

Let's check out the details of the subprocedures now. First, let's look at ASK:

Listing

```
TO ASK :QLIST
PR [ ]
(PR [WHAT'S THE CAPITAL OF] GPROP :QLIST "STATE [?])
END
```

Well, that's simple enough; GPROP outputs the name of a state found in the property list :QLIST. We use the same means to fetch the capital in GET.CAP:

Listing

```
TO GET.CAP :QLIST
OUTPUT GPROP :QLIST "CAPITAL
END
```

Here is the procedure that checks the answers:

Listing

```
TO COMPARE :ANS :RESPONSE
MAKE "GUESSES :GUESSES + 1
PR [ ]
TEST :ANS = :RESPONSE
IFT [PR [CORRECT YOU ARE!]]
IFT MAKE "RIGHT :RIGHT + 1
IFF [PR [SORRY, THAT'S WRONG.]]
IFF [(PR [THE CORRECT ANSWER IS] :ANS)]
PR [ ]
(PR [SO FAR THAT IS] :RIGHT [OUT OF] :GUESSES [RIGHT.])
END
```

Finally, CONTINUE? is the same procedure we used in MAKE.P.LIST. This completes QUIZZEM.

The last procedure in STATES is SUMUP. We've kept it pretty simple, but you can elaborate upon it if you wish.

Listing

```
TO SUMUP
LOCAL "PERCENT
MAKE "PERCENT 100 * :RIGHT / :GUESSES
IF :PERCENT = 100 [PR [TREMENDOUS! YOU KNOW YOUR
   CAPITALS!]]
IF AND NOT :PERCENT < 90 NOT :PERCENT = 100 [PR
   [EXCELLENT!]]
PR [ ]
(PR [YOU GOT] :RIGHT [OUT OF] :GUESSES [ATTEMPTS.])
(PR [THAT'S] :PERCENT [%.])
PR [I HOPE YOU ENJOYED THE GAME.]
END
```

That's the complete program. Fig. 15-2 shows a box chart for STATES. Here is a sample run:

Sample run

```
?TO STATES
```

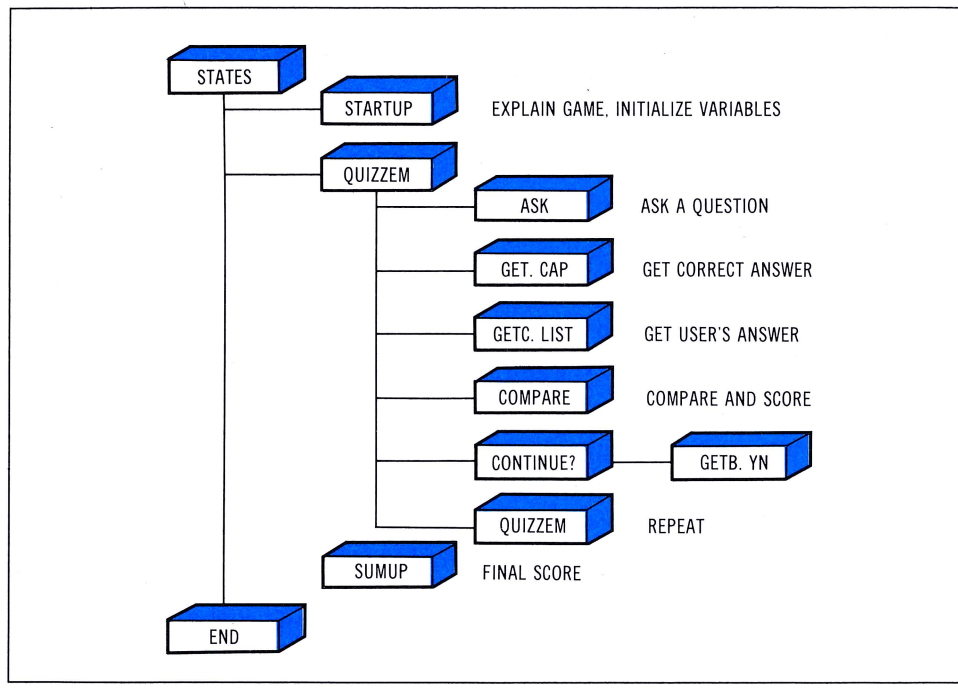


Fig. 15-2. Box chart for STATES.

```
DO YOU KNOW YOUR STATE CAPITALS?
LET'S FIND OUT. I'LL GIVE YOU THE
NAME OF A STATE, AND YOU GIVE ME THE
NAME OF ITS CAPITAL.

OK, HERE WE GO!


WHAT'S THE CAPITAL OF CALIFORNIA?
SACRAMENTO


CORRECT YOU ARE!


SO FAR THAT'S 1 OUT OF 1 RIGHT.
DO YOU WISH TO CONTINUE? Y OR N?
Y
```

```
WHAT'S THE CAPITAL OF VERMONT?
MOUNT VERNON

SORRY, THAT'S WRONG.
THE CORRECT ANSWER IS MONTPELIER

SO FAR THAT'S 1 OUT OF 2 RIGHT.
DO YOU WISH TO CONTINUE? Y OR N?
N

YOU GOT 1 RIGHT OUT OF 2 ATTEMPTS.
THAT'S 50 %.
I HOPE YOU ENJOYED THE GAME.
```

How embarrassing! We created the game and still missed a question. Well, that's life.

15.3 Extensions to the Game

We have produced a rather bare-bones form of the game. There are several extensions we can make, some simple, some more difficult. We'll discuss some possibilities now.

1. Avoid repeats. The nature of random selection is such that some states might pop up two or three times before another state is mentioned at all. The easy way to fix this is to keep a list of all the property lists called upon so far. Then, when a new property list is chosen, use MEMBERP to see if it has been used yet. If so, choose another. You could replace

```
MAKE "QLIST WORD "Q (RANDOM 50) + 1
```

with

```
MAKE "QLIST GET.QUEST
```

where GET.QUEST would be a new procedure that did the dirty work. It could be structured like GETB.NUMBER so that it outputs the property list name when successful and outputs itself when it fails.

You also should replace

```
IF NOT CONTINUE? [STOP]
```

with

```
IF OR :GUESSES = 50 NOT CONTINUE? [STOP]
```

so that the program stops when you run out of questions.

2. Switch questions. Give the capital and have the user guess the state. Create a new local variable "QTYPE in STATES. In SETUP ask the user whether he wants to guess capitals or states, perhaps by entering a one or a two, which gets assigned to "QTYPE. Then have a QUIZZEM1 and a QUIZZEM2. They would be identical except for switching the roles of states and capitals. You then could have a statement in STATES like

```
IF :QTYPE = 1 [QUIZZEM1] [QUIZZEM2]
```

to select which version is activated.

3. Give a second chance. In COMPARE, replace

```
IFF [(PR [THE CORRECT ANSWER IS] :ANS)]
```

with

```
IFF [SECONDCHANCE]
```

where SECONDCHANCE is a new procedure. In it, for example, we could print out the names of three capitals and say that the answer is one of them. If the user gets it right, increase :RIGHT by 0.5 (for partial credit). If wrong, reveal the true answer. This feature could be made optional via a menu selection in SETUP.

There are two tricky points to this extension. One is that we really should check to see that there are no duplicates in our list of three possible answers. That can be done placing the original answer in a list and adding new choices after they clear a NOT MEMBERP test. The second point is that we should print out the possibilities in random order. If the correct answer is, say, always last on the list, the user will catch on. Here is one approach. First, create the list of three answers, with the correct answer first. Then select one member at random from the following list:

```
[ [1 2 3] [2 3 1] [3 1 2]]
```

For example, [2 3 1] might be chosen. Then use recursion to use each element of this number list in turn as input for ITEM operating upon the answer list. For our example, we would get the second item, then the third, then the first from the answer list. This one's a bit of work; perhaps you can think of a better solution.

Those are some extensions you can work on. Meanwhile, let's look further at the property lists themselves.

15.4 Property Lists and Tables

Our 50 property lists ("Q1 through "Q50) hold information (data) about the 50 states. Each of these property lists can have more properties added to it. Here are some properties we could add: population, area, state bird, nickname, motto. For instance, since our "Q1 concerns Alabama, we could say

```
PPROP "Q1 "POPULATION 3917000
PPROP "Q1 "AREA 51609
PPROP "Q1 "BIRD [YELLOWHAMMER]
PPROP "Q1 "NICKNAME [HEART OF DIXIE]
PPROP "Q1 "MOTTO [WE DARE DEFEND OUR RIGHTS]
```

If we wish to pursue this project, we should modify our MAKE.P.LIST procedure:

Listing

```
TO ADD.DATA :X
LOCAL "QLIST
MAKE "QLIST WORD "Q :X
(PR [THE CURRENT STATE IS] GPROP :QLIST "STATE)
PR [POPULATION?]
PPROP :QLIST "POPULATION GETB.NUMBER
PR [AREA?]
PPROP :QLIST "AREA GETB.NUMBER
PR [STATE BIRD?]
PPROP :QLIST "BIRD GETB.LIST
RP [NICKNAME?]
PPROP :QLIST "NICKNAME GETB.LIST
PR [MOTTO?]
```

```
PPROP :QLIST "MOTTO GETB.LIST
IF NOT CONTINUE? [STOP]
ADD.DATA :X + 1
END
```

Once we enter all the information, we will have a hunk of information; such a hunk is what is called a data base. What can we do with it? One thing we could do is print a table. Here is a program to do just that:

Listing

```
TO TABLE
TEXTSCREEN CLEARTEXT SETCURSOR [0 0]
TYPE "STATE TAB 15 TYPE "POPULATION TAB 30 PR "AREA
PR [ ]
PTABLE 1
END


TO PTABLE :N
LOCAL "QLIST
MAKE "QLIST WORD "Q :N
TYPE GPROP :QLIST "STATE TAB 15
TYPE GPROP :QLIST "POPULATION TAB 30
PR GPROP :QLIST "AREA
IF :N = 50 [STOP]
PTABLE :N + 1
END


TO TAB :X
SETCURSOR LIST :X LAST CURSOR
END
```

TABLE clears the screen and prints a heading. The TAB procedure moves the cursor over so the columns are aligned. (TAB 15, for example, moves the cursor over to the 15th column.) The main work is done by PTABLE, which recursively prints the state name, population, and area for the 50 states. The beginning would look like this:

```
STATE                POPULATION          AREA
ALABAMA              3917000             51609
```

```
ALASKA          412000          586412
ARIZONA         2794000         113909
```

We could have printed more properties, but these three fit nicely on a 40-column screen. The output is a pleasant array of data. That reminds us of a connection between property lists and arrays.

15.5 Property Lists and Arrays

We can think of our set of 50 matched property lists as a two-dimensional array of data. Each property list is a row with 7 columns: the state name, capital, population, and so on. We then have 50 such rows. Fig. 15-3 illustrates this.

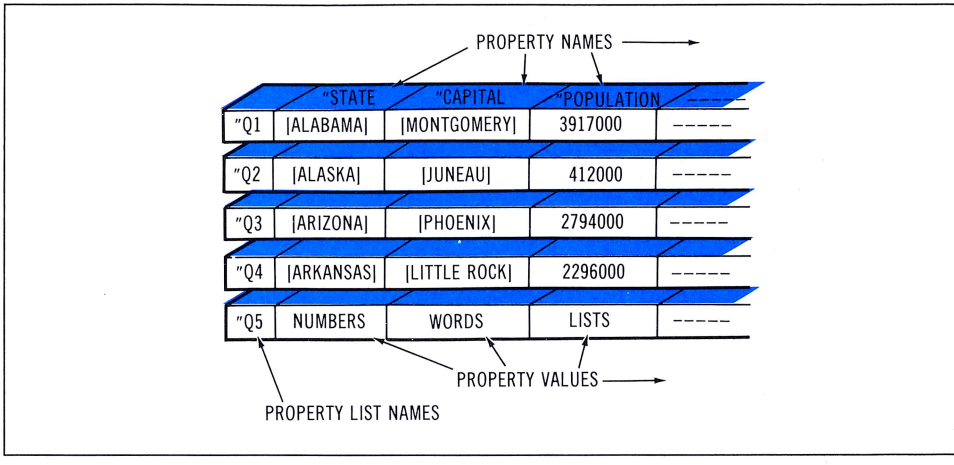


Fig. 15-3. A collection of rows of property lists.

This type of organization collects in one property list all the properties pertaining to one object (such as the state of Nevada). But the same data could be organized differently. Instead of 50 property lists, each with 7 properties, we could have used 7 property lists, each with 50 properties. For instance, we could have a property list called "POPULATIONS. The first property would be named "STATE1 and have the value of the population of the first state. Similarly, a property list "NAMES would have a property "STATE1 with the value [ALABAMA] and a "STATE2 with the value

[ALASKA]. Here all the data of one type (area, say) is in one property list. We can visualize the data of Fig. 15-3 organized now as columns of property lists instead of rows of property lists. See Fig. 15-4.

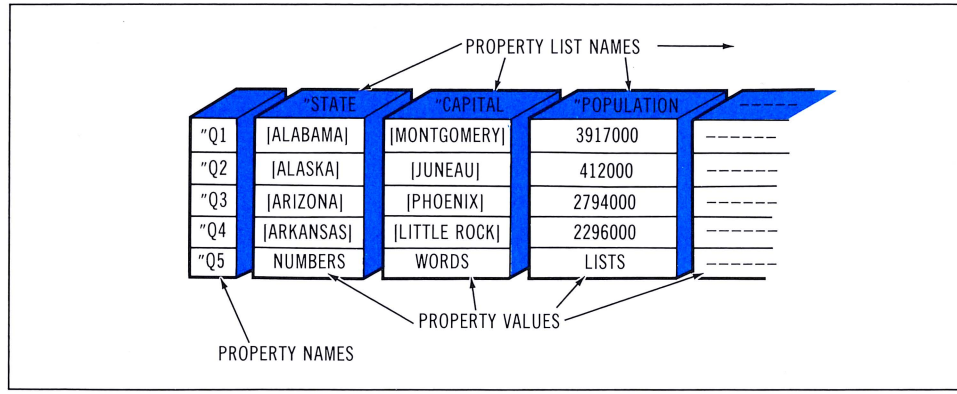


Fig. 15-4. A collection of columns of property lists.

Which is better? Both have their merits. The first approach (uniting different properties of the same object) ties together all the information about one object. This could be very useful if you want to, say, sort the states by population. The second approach (uniting the same property of different objects) lets you have, say, all the populations in one data object. This could be useful if, for instance, you wished to study the statistics of the population figures. In general, problems can be solved using either form. In some cases, one might be easier to work with, and in other cases it really makes no difference. For instance, *88 Apple Logo Programs* implements a generalization of the States and Capitals game using the second approach.

Okay, there is more than one way to use property lists to represent a data base. Let's return to some of the things we can use data bases.

15.6 Property Lists and Data Bases

Let's go back to our data base of state information. One thing we can do with this data base is to search it for some specific factor. Suppose we want a list of all those states with a population greater than five million. We can set up a program like this:

Listing

```
TO SEARCH :N :POP
```

```
LOCAL "QLIST
MAKE "QLIST WORD "Q :N
(PR [STATES WITH MORE THAN] :POP [PEOPLE:])
TEST GPROP :QLIST "POPULATION > :POP
IFT [(PR GPROP :QLIST "STATE [HAS] GPROP :QLIST "POPULATION
   [PEOPLE])]
SEARCH :N + 1 :POP
END
```

A sample run would begin like this:

Sample run

```
?SEARCH 1 5000000
STATES WITH MORE THAN 5000000 PEOPLE:
CALIFORNIA HAS 24196000 PEOPLE
FLORIDA HAS 10183000 PEOPLE
GEORGIA HAS 5574000 PEOPLE
```

and so on.

Another thing you could do with a data base is form sorted lists. For instance, you could rank the states by population, by area, or by the length of the state motto. We haven't discussed procedures for sorting, but you can find one, for example, in *88 Apple Logo Programs.* Suppose, for instance, we wish to sort the states by area. One approach would be to create the list [Q1 Q2 Q3 . . . Q49 Q50]. Then sort the list. But instead of comparing Q1 to Q2, compare GPROP "Q1 "AREA to GPROP "Q2 "AREA. If the second is larger, then rearrange the list to [Q2 Q1 . . . Q50], and so on, using the standard sorting techniques. In the end you get a list of property list names sorted by area. Suppose this list is called "AREALIST. Then you can print out the results with a procedure like this:

Listing

```
TO PR.PLIST :LIST
IF :LIST = [ ] [STOP]
TYPE GPROP FIRST :LIST "STATE
TAB 15
PR GPROP FIRST :LIST "AREA
PR.PLIST BUTFIRST :LIST
END
```

To use this on "AREALIST, type

```
PR.LIST :AREALIST
```

Here we use a second technique for processing several property lists. (The first, recall, was to use names like "Q1, "Q2, etc., and to use WORD "Q :N to recreate the names in numerical order.) This second technique consists of preparing a list of all the property list names. Use FIRST to extract the first property list name from the list, process that property list, shorten the input list with BUTFIRST, and continue the process. This method requires a little more effort to set up, but it can be used with any form of property list name.

Well, this barely touches on what you can do with property lists and data bases, but we hope it gives you a good idea of the possibilities and of some of the methods.

15.7 Summary

1. Property lists offer a powerful means of organizing related data.
2. A property list consists of a name for the property list, one or more names for particular properties, and values for those properties. This makes them rather similar to the records of Pascal and the structures of C. (FORTRAN and BASIC really have nothing that compare.)
3. We can create property lists with PPROP, and we can recover their contents with PPS, PLIST, and GPROP. To remove the contents, use REMPROP.

15.8 Review Questions and Exercises

Review Questions

1. How many property lists are created by the following sequence? How many properties?

```
PPROP "PET1 "SPECIES [CAT]
PPROP "PET1 "NAME [FAT NOSE]
PPROP "PET1 "AGE 3
```

2. What command will print FAT NOSE's age?

3. How would you remove the "AGE property from "PET1?

Exercises at the Terminal

1. Devise a property list format to describe a book. Devise another to describe a planet.
2. Construct a GET.QUEST procedure along the lines we suggested in 15.3.
3. The structure of STATES can be used for any quiz game involving matching a question to an answer. Think up one or more additional such games.
4. Write a utility called REM.ALL that can be used to remove all the values in a property list. REM.ALL could take one input, the name of the property list. (See "88 Apple Logo Programs.")

Answers to Review Questions

(1) One property list with three properties is created. (2) PR GPROP "PET1 "AGE (3) REMPROP "PET1 "AGE

APPENDIX A

SUMMARY OF LOGO COMMANDS

Notes: The key to using Logo primitives is to concentrate on inputs and outputs as indicated in the top line of each definition. A "#" sign on the left means that the primitive can take any number of inputs if parentheses are used.

Turtle Graphics

BACK
BK
One input (Distance). No outputs.
Moves turtle backward Distance steps.
Example: BACK 50

BACKGROUND
No inputs. One output.
Outputs a number representing the color of the background; 0 black; 1 white; 2 green; 3 violet; 4 orange; 5 blue; 6 black (for b&w TV).

CLEAN
No inputs. No outputs.
Erases the graphics screen without affecting the turtle (unlike CLEARSCREEN).

CLEARSCREEN
CS
No inputs. No outputs.
Erases screen; centralizes turtle; initializes heading to 0 (north).

DOT
One input (Position). No output.
Draws a dot of the current pen color at Position.
Example: DOT [100 0]
Places a dot halfway down the right edge of the screen.

FENCE
No inputs. No outputs.
Forces turtle to remain within the bounds of the screen.
Example: FENCE
FD 600
Gives error message "TURTLE OUT OF BOUNDS".

FORWARD
FD
One input (Distance). No outputs.
Moves turtle forward Distance steps.
Example: FORWARD 50.

HEADING	No inputs. One output. Outputs turtle's heading, with north 0, east 90, south 180, and west 270. **Example:** HEADING Outputs 45.
HIDETURTLE HT	No inputs. No outputs. Makes turtle invisible.
HOME	No inputs. No outputs. Moves turtle to center of screen and sets heading to 0.
LEFT LT	One input (Degrees). No outputs. Routes turtle Degrees left of present heading. **Example:** LEFT 45 Turns the turtle 45 degrees left.
PEN	No inputs. One output (List). Outputs a two-word list. The first member is PENDOWN, PENERASE, PENUP, or PENREVERSE. The second member is the color number. **Example:** PEN Outputs [PENDOWN 2].
PENCOLOR PC	No inputs. One output (Number). Outputs number representing current color; 0 black; 1 white; 2 green; 3 violet; 4 orange; 5 blue.
PENDOWN PD	No inputs. No outputs. Puts the turtle's pen down so it draws when moved.
PENERASE PE	No inputs. No outputs. Puts the turtle's eraser down so that it erases when moved.
PENREVERSE PX	No inputs. No outputs. Interchanges pen color and background color, drawing where there aren't lines and erases where there are.
PENUP PU	No inputs. No outputs. Lifts the pen up so no lines are drawn where the turtle moves.
POS	No inputs. One output (List). Outputs the coordinates of the current position of the turtle in the form [x y].

RIGHT
RT
One input (Degrees). No outputs.
Turns the turtle right (clockwise) Degrees.
RIGHT 30
Turns the turtle 30 degrees right.

SCRUNCH
No inputs. One output.
Outputs the aspect ratio, the ratio of vertical turtle step size to the horizontal one.

SETBG
One input (Colornumber). No outputs.
Sets background color corresponding to Colornumber.

SETHEADING
SETH
One input (Degrees). No outputs.
Turns the turtle so it is heading in the direction Degrees.
Example: SETHEADING −45
Heads the turtle northwest.

SETPC
One input (Colornumber). No outputs.
Sets the color of the pen to Colornumber: 0 black; 1 white; 2 green; 3 violet; 4 orange; 5 blue.

SETPEN
One input (Pair). No outputs.
Sets pen state to Pair. The first word is PENDOWN, PENERASE, PENUP, or PENREVERSE. The second word is the color number.
Example: SETPEN [PENREVERSE 5]
Has the same effect as PENREVERSE SETPC 5.

SETPOS
One input (Position). No outputs.
Moves the turtle to Position.
Example: SETPOS [100 0]
Moves the turtle halfway down right edge of the screen.

SETSCRUNCH
One input (Ratio). No outputs.
Sets the aspect ratio to Ratio; changes YCOR accordingly.
Example: SETSCRUNCH .75
Makes vertical turtle step ¾ the length of horizontal one.

SETX
One input (X). No outputs.
Moves the turtle horizontally to x-coordinate X.
Example: SETX 100
Moves the turtle horizontally to right edge of screen.

SETY
One input (Y). No outputs.
Moves turtle vertically to y-coordinate Y.
Example: SETY −100
Moves the turtle vertically to lower edge of screen.

SHOWNP
No inputs. One output (Boolean).
Outputs TRUE if turtle is shown; FALSE otherwise.

SHOWTURTLE
ST
No inputs. No outputs.
Makes turtle visible.

TOWARDS
One input (Position). One output (Heading).
Outputs Heading to make turtle face Position.
Example: SETHEADING TOWARDS [20 10]
Heads the turtle in the direction of position [20 10].

WINDOW
No inputs. No outputs.
Makes the turtle field unbounded.
Example:
```
WINDOW
FD 600
PRINT POS
30.015 570.105
```

WRAP
No inputs. No outputs.
Makes turtle field wrap around the edges of screen.
Example:
```
WRAP
FD 700
PRINT POS
30.015 20.975
```

XCOR
No inputs. One output.
Outputs the x-coordinate of current position of turtle.

YCOR
No inputs. One output.
Outputs the y-coordinate of current position of turtle.

Words and Lists

ASCII
One input (character). One output.
Outputs the ASCII code for character.
See Appendix B for the full ASCII character set.
Example: ASCII "D
Outputs 68.

BUTFIRST
BF
One input (Object). One output.
Outputs all but the first element of Object.
Example: BUTFIRST [NOW IS THE TIME]
Outputs IS THE TIME.

BUTLAST
BL
One input (Object). One output.
Outputs all but the last element of Object.
Example: BUTLAST [NOW IS THE TIME]
Outputs NOW IS THE.

CHAR
One input (N). One output.
Outputs the character whose ASCII code is N.
Example: CHAR 65
Outputs A.

COUNT
One input (List). One output.
Outputs the number of elements in List.
Example: COUNT [HOW MANY ARE THERE?]
Outputs 4.

EMPTYP
One input (Object). One output.
Outputs TRUE if Object is the empty word or the empty list; FALSE otherwise.
Example: EMPTYP 7
Outputs FALSE.

EQUALP
Two inputs (Object1, Object2). One output.
Outputs TRUE if Object1 and Object2 are equal; FALSE otherwise.
Example: EQUALP 5 2+3
Outputs TRUE.

FIRST
One input (Object). One output.
Outputs the first element of Object.
Example: FIRST [A B C]
Outputs A.

FPUT
Two inputs (Object, List). One output.
Outputs a new list formed by putting Object at the beginning of the List.
Example: FPUT "RAT [BAT HAT]
Outputs [RAT BAT HAT].

ITEM
Two inputs (N, List). One output.
Outputs the Nth element of List.
Example: ITEM 2 [ROME CHICAGO MOSCOW]
Outputs CHICAGO.

LAST
One input (Object). One output.
Outputs the last element of Object.
Example: LAST [CAT COW HEN DOG]
Outputs DOG.

LIST	Two inputs (Object1, Object2). Outputs a list whose elements are Object1 and Object2. **Example:** LIST "HOUSE [BARN CABIN] Outputs [HOUSE [BARN CABIN]].
#	(LIST Object 1 Object2 Object3 . . .)
LISTP	One input (Object). No output Outputs TRUE if Object is a list, FALSE otherwise **Example:** LISTP [HERE IS A LIST] Outputs TRUE.
LPUT	Two inputs (Object, List). One output. Outputs a new list formed by putting Object at the end of the List. **Example:** LPUT "123 [BAT HAT] Outputs [BAT HAT 123].
MEMBERP	Two inputs (Item, List). One output. Outputs TRUE if Item is a member of List, False otherwise. Item may be any data object. **Example:** MEMBERP [1 2] [[A B] [1 2] [HI THERE]] Outputs TRUE.
NUMBERP	One input (Object). One output. Outputs TRUE if Object is a number; FALSE otherwise. **Example:** NUMBERP 5 Outputs TRUE.
SENTENCE SE	Two inputs (Object1, Object2). One output (List). Outputs a list made up of the words in Object1 and Object2. **Example:** SENTENCE [NOW IS] [THE TIME] Outputs [NOW IS THE TIME].
#	(SENTENCE Object1, Object2, Object3, . . .)
WORD	Two inputs (Word1, Word2). Outputs a word made up of Word1 and Word2. **Example:** WORD "BAT "MAN Outputs BATMAN.
#	(WORD Word1, Word2, Word3, . . .)
WORDP	One input (Object). One output. Outputs TRUE if Object is a word; FALSE otherwise. **Example:** WORDP "XXX Outputs TRUE.

Variables

LOCAL
One input (Name). No outputs.
Makes its inputs local to the procedure where LOCAL occurs.

#
(Local Name1, Name2, Name3, . . .)

MAKE
Two inputs (Name, Object). No output.
Gives the variable Name the value Object.
Example: MAKE "THIS 475
PR :THIS
475

NAME
Two inputs (Name, Object). No outputs.
Gives variable Name the value Object.
Example: NAME "MAN "JOHN
PR :MAN
JOHN.

NAMEP
One input (Name). One output.
Outputs TRUE if name has a value; FALSE otherwise.
PR NAMEP "HOUSE
Outputs FALSE if HOUSE has not been assigned a value.

THING
One input (Name). One output.
Outputs the value of the variable Name.
Example: MAKE "SIZE 33
THING :SIZE
Outputs 33.

Arithmetic Operations

ARCTAN
One input (N). One output.
Outputs the arctangent (inverse tangent) of N.
Example: ARCTAN 1
Outputs 45.

COS
One input (N). One output.
Outputs the cosine of N degrees.
Example: COS 60
Outputs .5.

INT	One input (X). One output. Outputs the integer portion of X (truncates). **Example:** INT 6.7 Outputs 6.
PRODUCT	Two inputs (N1, N2). One output. Outputs the product of its inputs. **Example:** PRODUCT 8 4 Outputs 32.
#	(Product N1, N2, N2, . . .).
QUOTIENT	Two inputs (X1, X2). One output. Outputs X1 divided by X2, truncated to an integer. **Example:** QUOTIENT 18 4 Outputs 4.
RANDOM	One input (N). One output. Outputs a random non-negative integer less than N. **Example:** RANDOM 6 Outputs 0, 1, 3, 4, or 5.
REMAINDER	Two inputs (X1, X2). One output. Outputs the remainder obtained when X1 is divided by X2. **Example:** REMAINDER 17 5 Outputs 2.
RERANDOM	Makes RANDOM produce the same sequences each time.
ROUND	One input (N). One output. Outputs N rounded to the nearest integer. **Example:** ROUND 5.501 Outputs 6.
SIN	One input (N). One output. Outputs the sine of N degrees. **Example:** Sin 30 Outputs .5.
SQRT	One input (N). One output. Outputs the square root of N, or error message if N is negative. **Example:** SQRT 36 Outputs 6.
SUM	Two inputs (X1, X2). One output.

Outputs the sum of X1 and X2.
Example: SUM 6 4
Outputs 10.

\# (SUM X1, X2, X3, . . .)

\+ Two inputs (X1, X2). One output.
Outputs the sum of its inputs.
Example: 4+5, 4 + 5, 3+3+3
All output 9.

− Two inputs (X1, X2). One output.
Outputs the result of X1−X2
Example: 5−3, 5 − 3, 5− 3
All output 2.

− One input (X1). One output.
Outputs the negative of X1.
Example: −5
Outputs −5.

* Two inputs (X1, X2). One output.
Outputs the product of its inputs.
Example: 5*6, 5 * 6
Both output 30.

/ Two inputs (X1, X2). One output.
Outputs X1 divided by X2.
Example: 8/4, 8 / 4
Both output 2.

> Two inputs (X1, X2). One output.
Outputs "TRUE if X1 is greater than X2, "FALSE otherwise.
Example: 5>4, 5 > 4
Both output "TRUE.

< Two inputs (X1, X2). One output.
Outputs "TRUE if X1 is less than X2, "FALSE otherwise.
Example: 4<5, 4 < 5
Both output "TRUE.

= Two inputs (Object1, Object2). One output.
Outputs "TRUE if Object1 equals Object2, "FALSE otherwise.
Example: 4=4, 4 = 4, (FIRST "OTHELLO) = "O
All output "TRUE.

Conditionals and Flow of Control

CO

One input (Object – optional). No outputs.
Resumes running a procedure after a PAUSE or CTRL-Z. If CO has an input, it becomes the output from PAUSE.

IF

Inputs: (Pred, Instructionlist1, Instructionlist2 (optional)).
Outputs—whatever instruction list outputs (if any). If Pred is TRUE, runs Instructionlist1. If Pred is FALSE, runs Instructionlist2.
Example: IF (FIRST :NAMES) = "MITCH [PR [HI MITCH]]
[PR [NOT HERE]]
Prints HI MITCH if condition is met.

IFFALSE
IF

One input (Instructionlist). Outputs: same as Instructionlist, if any.
Runs Instructionlist if result of most recent TEST was FALSE.

IFTRUE
IFT

One input (Instructionlist). Outputs: same as Instructionlist, if any.
Runs Instructionlist if result of most recent TEST was TRUE.
Example: TEST :X > 5
IFT [PR [THE NUMBER IS GREATER THAN 5]]
IFF [PR [GIVE ME A BIGGER NUMBER]]

OUTPUT
OP

One input (Object). One output (Object).
Makes Object the output of current procedure, and returns control to the caller.
Example: OUTPUT :SIZE
Outputs the value of SIZE, which must now be used as input.

PAUSE

No inputs. One output (Message).
Suspends current procedure and outputs Message.

REPEAT

Two inputs (N, Instructionlist). Same outputs as Instructionlist, if any.
Runs Instructionlist N times.
Example: REPEAT 3 [FD 30 RT 120]
Draws a triangle.

RUN — One input (Instructionlist). Same outputs as Instructionlist, if any.
Runs Instructionlist as if typed directly.
Example: MAKE "COMMANDS [REPEAT 6 [FD 50 RT 60]
RUN :COMMANDS
Draws a hexagon.

STOP — No inputs. No outputs.
Stops the current procedure and returns control to the caller.

TEST — One input (Pred). No outputs.
Stores value of Pred (TRUE or FALSE) for subsequent tests.
Example: See IFTRUE or IFFALSE.

CTRL-G — No inputs. No output.
Stops current procedure, returns Logo to top level, and prints a ?.

CTRL-W — No inputs. No outputs.
Interrupts whatever is running. Typing any other character resumes what was running.

CTRL-Z — No inputs. No outputs.
Interrupts whatever is running, causing a PAUSE.

Logical Operations

AND — Two inputs (Pred1, Pred2). One output (TRUE or FALSE).
Outputs TRUE if Pred1 and Pred2 are true; FALSE otherwise.

\# — (AND Pred1, Pred2, Pred3, . . .).
Example: AND :X>5 :X<10
Outputs TRUE if X is between 5 and 10.

NOT — One input (Pred). One output (TRUE or FALSE).
Outputs TRUE if Pred is FALSE; TRUE otherwise.
Example: NOT EQUAL "ABC "123
Outputs TRUE.

OR — Two inputs (Pred1, Pred2). One output (TRUE or FALSE).
Outputs FALSE if both inputs are false. TRUE otherwise.

\# — (OR Pred1, Pred2, Pred3 . . .)
Example: OR :X<5 :X>10
Outputs "TRUE if X is outside of 5-10.

The Outside World

BUTTONP
One input (Paddlenumber). One output (TRUE or FALSE).
Outputs TRUE if button on Paddlenumber is down; FALSE otherwise.

KEYP
No inputs. One output (TRUE or FALSE).
Outputs TRUE if there is at least one character waiting to be read; FALSE otherwise.

PADDLE
One input (Paddlenumber). One output.
Outputs number between 0 and 255 representing the rotation of the dial on the specified paddle.

PRINT
PR
#
One input (Object). No outputs.
Prints its input on the screen, followed by a return.
(PRINT Object1 Object2 . . .).
Example: (PR [THE NUMBER IS] :X)
Prints THE NUMBER IS 7 if the value of X is 7.

READCHAR
RC
No. inputs. One output.
Outputs the first character typed. Waits for character.
Example: MAKE "ANS RC

READLIST
RL
No inputs. One output.
Waits for the user to type a line, and outputs in list form.
Example: MAKE "ANS RL

SHOW
One input (Object). No output.
Prints Object on the screen, followed by a return.

TYPE
#
One input (Object). No output.
Prints its input on the screen without a return.
(TYPE Object1 Object2)
Example: TYPE [THE PRODUCT OF X*Y IS] PRINT :X*:Y

WAIT
One input (Number). No outputs.
Causes Logo to pause for number/60 seconds.
WAIT 120

Text and Screen Commands

CLEARTEXT
No inputs. No outputs.
Clears text screen and initializes cursor position.

CURSOR	No inputs. One output. Outputs list of column and line numbers of cursor position.
FULLSCREEN	Devotes the screen to graphics, with turtle field showing.
SETCURSOR	One input (Position). No output. Sets the cursor to Position (Lines 0-23, cols. 0-39). **Example:** SETCURSOR [0 12] Puts cursor half-way down the left edge of the screen.
SPLITSCREEN	Devotes top 20 lines of screen to turtle field.
TEXTSCREEN	Devotes the entire screen to text.
CTRL-L	Devotes the entire screen to graphics.
CTRL-S	Devotes the top 20 lines of the screen to graphics.
CTRL-T	Devotes the entire screen to text.

Workspace Management

BURY	One input (Packagename). No outputs. Buries all procedures and names in Package.
ERALL	Optional input (Package or Packagelist). No output. Erases all unburied procedures and variables (in Package or Packagelist) from the workspace.
ERASE	One input (Name or Namelist). No outputs. Erases the named procedure(s) from the workspace. **Example:** ERASE [CIRCLE PENT] Erases the CIRCLE and PENT procedures.
ERN	One input (Name or Namelist). No outputs. Erases the named variable(s) from workspace. **Example:** ERN [V1 V2] Erases V1 and V2 variables.
ERNS	Optional input (Package or Packagelist). No outputs. Erases all variables (in Package or Packagelist) from workspace.
ERPS	Optional input (Package or Packagelist). Erases all procedures (in Package or Packagelist) from workspace.

PACKAGE	Two inputs (Package, Name or Namelist). No outputs. Puts each named procedure in Package. **Example:** PACKAGE "ANIMAL [CAT DOG] Puts CAT and DOG in the package ANIMAL.
PKGALL	One input (Package). No outputs. Puts into Package all procedures and variables that are not already in packages.
PO	One input (Name or Namelist). No outputs. Prints the definitions of the named procedure(s).
POALL	Optional input (Package or Packagelist). Prints definition of every procedure and value of every variable (in Package or Packagelist).
PONS	Optional input (Package or Packagelist). No outputs. Prints the name and value of every variable (in Package or Packagelist).
POPS	Optional input (Package or Packagelist). No outputs. Prints the definition of every procedure in Package or Packagelist.
POTS	Optional input (Package or Packagelist). No outputs. Prints the title line of every procedure (in Package or Packagelist).
UNBURY	One input (Package). No outputs. Unburies all procedures and names in Package.

Files

CATALOG	No inputs. No outputs. Prints the names of all files on the disk.
DISK	No inputs. One output. Outputs list of 3 numbers: the disk drive number, the slot number of the disk drive, and the volume number of the disk most recently used for CATALOG or set with SETDISK.
ERASEFILE	One input (File). No outputs. Erases File named FILE.LOGO from disk. **Example:** ?ERASEFILE "NOTES Erases file named NOTES.LOGO.

LOAD — One or two inputs (File, Package (optional)). No outputs. Loads the contents of FILE.LOGO into the workspace. If Package is input, it specifies the package everything goes into.
Example: LOAD "SQUARE
Loads the file named SQUARE.LOGO from the disk to the workspace

SAVE — One or two inputs (File, Package (optional)). No outputs. Creates a file named FILE.LOGO and saves all procedures and variables in Package if this is input, and of the entire workspace otherwise.
Example: SAVE "SQUARE
Saves all the workspace contents in a file called SQUARE.LOGO.

SETDISK — One to three inputs (Drive, Slot, Volume (last 2 optional). No outputs.
Sets the disk drive to Drive, slot to Slot, and volume to Volume.

Advanced Primitives or Rarely Used Primitive

GPROP — Two inputs (Name, Prop). One output.
Outputs the value of Prop property of Name.
Example: GROUP "CITY "A4
Outputs SAN FRANCISCO (see below PPROP for reason)

PLIST — One input (Name). One output.
Outputs the property list associated with Name.

PPROP — Three inputs (Name, Prop, Object). No outputs.
Gives Name the property Prop with value Object.
Example: PPROP "CITY "A4 [SAN FRANCISCO]
Gives the value of SAN FRANCISCO to the name A4 in the property list CITY.

PPS — Optional input (Package or Packagelist). Variable outputs.
Prints the property list(s) of everything (in Package or Packagelist) in workspace.

REMPROP

Two inputs (Name, Prop). No outputs.
Removes property pair Prop from the property list of Name.
Example: REMPROP "CITY "A4
Removes the name A4 and its value from the property list CITY.

CATCH

Two inputs (Name, Instructionlist). Same outputs as Instructionlist.
Runs Instructionlist. If a THROW Name is called while Instructionlist is run, control returns to the CATCH.

ERROR

No inputs. One output.
Outputs six-element list containing information about the most recent error which has not had a message printed or output by ERROR.

THROW

One input (Name). No outputs.
Used to match name with CATCH name—error if none exists.

COPYDEF

Two inputs (Newname, Name). No outputs.
Copies the definition of Name to that for Newname.
Example: COPYDEF "F "FORWARD
Gives F the same definition as FORWARD.

DEFINE

Two inputs (Name, List). No outputs.
Makes List the definition of the procedure Name.
First list in List must be the inputs to Name.
Example: DEFINE "BOX [[SIZE] [REPEAT 4 [FD :SIZE RT 90]]].

PRIMITIVEP

One input (Name). One output.
Outputs TRUE if Name is name of primitive; FALSE otherwise.
Example: PRIMITIVEP "FORWARD
Outputs TRUE.

TEXT

One input (Name). One output.
Outputs the definition of Name as a list of lists.

GO

One input (Name). No outputs.
Used to loop back to instruction following LABEL.
Example:
```
TO COUNTUP
   MAKE "X 1
   LABEL "LOOP
   PRINT :X
   MAKE "X :X+1
```

```
GO "LOOP
END
```

It is better to use recursion than to use GO like this.

LABEL — One input (Name). No outputs.
Used to pass control to instruction using GO.

NODES — No inputs. One output.
Outputs the number of free nodes.

RECYCLE — No inputs. No outputs.
Frees as many nodes as possible through a garbage collection.

REPARSE — Makes reparsing happen immediately.

.BPT — No input. No output.
Enters the Apple monitor. Logo may be resumed by typing 803G followed by RETURN.

.CONTENTS — Outputs list of currently defined objects, including variables, procedures, Logo primitives.

.DEPOSIT — Two inputs (N,A). No outputs.
Writes A into machine address N.

.EXAMINE — One input (Object). One output.
Outputs the contents of machine address Object (decimal) if Object is a number. Otherwise, outputs the address of the first node of Object.

.PRINTER — One input (N). No outputs.
Message to Logo that any subsequent information displayed on the screen is to be printed.
Example: .PRINTER 1
If printer card is in slot #1, sends text to printer.
Use .PRINTER 0 to turn off printer and to return to screen.

Control Characters

CTRL-A — Moves cursor to beginning of line.

CTRL-B — Moves cursor back one character.

CTRL-C — Exits editor, saving changes made.

CTRL-D — Deletes character under cursor.

CTRL-E	Moves cursor to end of line.
CTRL-F	Moves cursor forward one character.
CTRL-G	Emergency STOP, aborts procedure or editing.
CTRL-H	Same as left arrow.
CTRL-K	Kills all text to right of cursor.
CTRL-L	Scrolls screen so current line is at center. (editor). Same as FULLSCREEN (command level).
CTRL-M	Same as RETURN.
CTRL-N	Moves cursor down one line.
CTRL-O	Opens a new line at position of cursor.
CTRL-P	Moves cursor up one line.
CTRL-Q	Allows insertion of delimiting characters.
CTRL-S	Same as SPLITSCREEN.
CTRL-T	Same as TEXTSCREEN.
CTRL-U	Same as right arrow.
CTRL-V	Scrolls cursor down text one page.
CTRL-W	Temporary stop. Any other key resumes program. Helpful to stop screen scroll.
CTRL-Y	Inserts previous line erased with CTRL-K. Useful for moving a line in the editor.
CTRL-Z	Same as PAUSE, except that used during a program.
ESC >	Moves cursor to end of editor buffer.
ESC <	Moves cursor to beginning of editor buffer.

Editing Primitives

EDIT	One input (Name, Namelist). No output.
ED	Starts up Logo editor. Use CTRL-C to leave editor, saving changes. **Example:** ED "BOX

EDNS	Zero or one input (Package, Package list). No outputs. With no input, puts all variables in editor. With Package input, puts all variables in package in editor. With Package list as input, puts all variables in list in editor.
TO	One input (Name—unquoted). No output. Enters procedure editor. Use END to complete procedure definition. **Example:** TO BOX

APPENDIX B
ASCII CHART

DECIMAL-HEXADECIMAL-OCTAL-BINARY-ASCII NUMERICAL CONVERSION

DEC X_{10}	HEX X_{16}	OCT X_8	Binary P X_2	ASCII	Key*
0	00	00	0 000 0000	NUL	CTRL/1
1	01	01	1 000 0001	SOH	CTRL/A
2	02	02	1 000 0010	STX	CTRL/B
3	03	03	0 000 0011	ETX	CTRL/C
4	04	04	1 000 0100	EOT	CTRL/D
5	05	05	0 000 0101	ENQ	CTRL/E
6	06	06	0 000 0110	ACK	CTRL/F
7	07	07	1 000 0111	BEL	CTRL/G
8	08	10	1 000 1000	BS	CTRL/H, BACKSPACE
9	09	11	0 000 1001	HT	CTRL/I, TAB
10	0A	12	0 000 1010	LF	CTRL/J, LINE FEED
11	0B	13	1 000 1011	VT	CTRL/K
12	0C	14	0 000 1100	FF	CTRL/L
13	0D	15	1 000 1101	CR	CTRL/M, RETURN
14	0E	16	1 000 1110	SO	CTRL/N
15	0F	17	0 000 1111	SI	CTRL/O
16	10	20	1 001 0000	DLE	CTRL/P
17	11	21	0 001 0001	DC1	CTRL/Q
18	12	22	0 001 0010	DC2	CTRL/R
19	13	23	1 001 0011	DC3	CTRL/S
20	14	24	0 001 0100	DC4	CTRL/T
21	15	25	1 001 0101	NAK	CTRL/U
22	16	26	1 001 0110	SYN	CTRL/V
23	17	27	0 001 0111	ETB	CTRL/W
24	18	30	0 001 1000	CAN	CTRL/X
25	19	31	1 001 1001	EM	CTRL/Y
26	1A	32	1 001 1010	SUB	CTRL/Z
27	1B	33	0 001 1011	ESC	ESC, ESCAPE
28	1C	34	1 001 1100	FS	CTRL <
29	1D	35	0 001 1101	GS	CTRL/
30	1E	36	0 001 1110	RS	CTRL/=

DEC X_{10}	HEX X_{16}	OCT X_8	Binary P X_2	ASCII	Key*
31	1F	37	1 001 1111	US	CTRL/-
32	20	40	1 010 0000	SP	SPACEBAR
33	21	41	0 010 0001	!	!
34	22	42	0 010 0010	”	”
35	23	43	1 010 0011	#	#
36	24	44	0 010 0100	$	$
37	25	45	1 010 0101	½	½
38	26	46	1 010 0110	&	&
39	27	47	0 010 0111	‘	‘
40	28	50	0 010 1000	(	(
41	29	51	1 010 1001	)	)
42	2A	52	1 010 1010	*	*
43	2B	53	0 010 1011	+	+
44	2C	54	1 010 1100	’	’
45	2D	55	0 010 1101	-	-
46	2E	56	0 010 1110	.	.
47	2F	57	1 010 1111	/	/
48	30	60	0 011 0000	0	0
49	31	61	1 011 0001	1	1
50	32	62	1 011 0010	2	2
51	33	63	0 011 0011	3	3
52	34	64	1 011 0100	4	4
53	35	65	0 011 0101	5	5
54	36	66	0 011 0110	6	6
55	37	67	1 011 0111	7	7
56	38	70	1 011 1000	8	8
57	39	71	0 011 1001	9	9
58	3A	72	0 011 1010	:	:
59	3B	73	1 011 1011	;	;
60	3C	74	0 011 1100	<	<
61	3D	75	1 011 1101	=	=
62	3E	76	1 011 1110	>	>
63	3F	77	0 011 1111	?	?
64	40	100	1 100 0000	@	@
65	41	101	0 100 0001	A	A
66	42	102	0 100 0010	B	B
67	43	103	1 100 0011	C	C
68	44	104	0 100 0100	D	D
69	45	105	1 100 0101	E	E
70	46	106	1 100 0110	F	F

DEC X_{10}	HEX X_{16}	OCT X_8	Binary P X_2	ASCII	Key*
71	47	107	0 100 0111	G	G
72	48	110	0 100 1000	H	H
73	49	111	1 100 1001	I	I
74	4A	112	1 100 1010	J	J
75	4B	113	1 100 1011	K	K
76	4C	114	1 100 1100	L	L
77	4D	115	0 100 1101	M	M
78	4E	116	0 100 1110	N	N
79	4F	117	1 100 1111	O	O
80	50	120	0 101 0000	P	P
81	51	121	1 101 0001	Q	Q
82	52	122	1 101 0010	R	R
83	53	123	0 101 0011	S	S
84	54	124	1 101 0100	T	T
85	55	125	0 101 0101	U	U
86	56	126	0 101 0110	V	V
87	57	127	1 101 0111	W	W
88	58	130	1 101 1000	X	X
89	59	131	0 101 1001	Y	Y
90	5A	132	0 101 1010	Z	Z
91	5B	133	1 101 1011	[	[
92	5C	134	0 101 1100	/	/
93	5D	135	1 101 1101	]	]
94	5E	136	1 101 1110	^	^
95	5F	137	0 101 1111	–	–
96	60	140	0 110 0000	‘	‘
97	61	141	1 110 0001	a	a
98	62	142	1 110 0010	b	b
99	63	143	0 110 0011	c	c
100	64	144	1 110 0100	d	d
101	65	145	0 110 0101	e	e
102	66	146	0 110 0110	f	f
103	67	147	1 110 0111	g	g
104	68	150	1 110 1000	h	h
105	69	151	0 110 1001	i	i
106	6A	152	0 110 1010	j	j
107	6B	153	1 110 1011	k	k
108	6C	154	0 110 1100	l	l
109	6D	155	1 110 1101	m	m
110	6E	156	1 110 1110	n	n

DEC X_{10}	HEX X_{16}	OCT X_8	Binary P X_2	ASCII	Key*
111	6F	157	0 110 1111	o	o
112	70	160	1 111 0000	p	p
113	71	161	0 111 0001	q	q
114	72	162	0 111 0010	r	r
115	73	163	1 111 0011	s	s
116	74	164	0 111 0100	t	t
117	75	165	1 111 0101	u	u
118	76	166	1 111 0110	v	v
119	77	167	0 111 0111	w	w
120	78	170	0 111 1000	x	x
121	79	171	1 111 1001	y	y
122	7A	172	1 111 1010	z	z
123	7B	173	0 111 1011	R	R
124	7C	174	1 111 1100	/	/
125	7D	175	0 111 1101	T	T
126	7E	176	0 111 1110	~	~
127	7F	177	1 111 1111	DEL	DEL, RUBOUT

P = Parity bit: "1" for odd number of 1's, "0" for even number of 1's.

*Those key sequences consisting of "CTRL/" are typed in by simultaneously pressing the CTRL key and the key indicated. These sequences are based on those defined for the Diablo 1640 keyboard. These key sequences may be defined differently on other keyboards.

ASCII Definitions

ACK (Acknowledgment)—Used as a general "yes" answer to various queries, but it also sometimes indicates "I received your last transmission and I'm ready for your next."

ASCII—American Standard Code for Information Interchange.

BEL (Bell)—Activates a bell, beeper, or other audible alarm.

BS (Backspace)—Moves the carriage, print head, or cursor back one space or position.

CAN (Cancel)—Indicates that the material in the previous transmission is to be disregarded. The amount of material is decided by the user.

CR (Carriage Return, or Return)—Moves carriage, print head, or cursor back to beginning of line. On most terminals, the RETURN key causes both a CR and an FF (Form Feed).

DC1-DC4 (Device Controls)—These are used to control the user's terminal or similar devices. There are no standard functions assigned, except that DC4 frequently means "stop." The CCITT suggests a number of possible assignments; in general, they prefer using the first two controls for "on," and the last two for "off," and DC2 and DC4 to refer to the more important device. In some systems, these codes are labeled XON, TAPE, XOFF, and $\overline{\text{TAPE}}$, respectively. X means "transmitter," and TAPE and $\overline{\text{TAPE}}$ means "tape on" and "tape off." These labels are found on the keytops of some terminals.

DEL (Delete)—Used to delete a character. It is called RUBOUT on some terminals. It is not strictly a control character since it is not grouped with the other ASCII control characters. The DEL function has a binary all-ones bit pattern ($1111\ 1111_2$), and the reason for this is historic: the only way to erase a bit pattern punched into paper tape was to punch out all the holes so that the resulting pattern was equivalent to a null. ASCII still considers DEL equivalent to a null, although many operating systems use it to erase the preceding character.

DLE (Data Link Escape)—This control function uses a special type of escape sequence specifically used for controlling the data line and transmission facilities.

EM (End of Medium)—This is used to indicate the end of paper tape (or other storage medium) or that this is the end of the material on the medium.

ENQ (Enquiry)—This is usually used for requesting identification or status information. In some systems, this code is called WRU—"Who are you?"

EOT (End of Transmission)—Code used to mark end of transmission after one or more messages.

ESC (Escape)—This code marks the beginning of an escape sequence. An escape sequence consists of a series of codes which, as a group, have a special meaning, usually a control function. On some terminals, ESC is called ALT MODE.

ETB (End of Transmission Block)—This code is used when it is desired to break up a long message into blocks. ETB is used to mark block boundaries. The blocks usually have nothing to do with the format of the message being transmitted.

ETX (End of Text)—This code is used to mark the end of a text. See SOH. This code was originally called EOM, "End of Message," and may be labeled as such on some terminals.

FF (Form Feed)—Advance to the top of next page.

FS, GS, RS, US (File, Group, Record and Unit Separator)—These are a set of codes that are used as "information separators" for delimiting portions of information. There is no standard usage, except that FS is expected to refer to the largest division, and US to the smallest.

HT (Horizontal Tab)—Used to tab carriage, print wheel, or cursor to the next predetermined stop on the same line. It is usually up to the user to decide where the horizontal tab stops are to be positioned.

LF (Line Feed)—This code moves the carriage, print head, or cursor down one line. Most systems combine CR (carriage return) with LF, and a new line is then called new line (NL).

NAK (Negative Acknowledgment)—Used to indicate "no" in answer to various queries. It is sometimes defined as "I received your last transmission, but it had errors and I'm awaiting a retransmission."

NUL (Null)—This code is used mainly as a space filler. See also SYN.

SI (Shift IN)—Used after an SO code to indicate that codes revert to normal ASCII meaning.

SO (Shift Out)—Indicates that bit patterns to follow will have meanings outside the standard ASCII set and will continue to do so until SI is entered.

SOH (Start Of Heading)—This code is used to mark the beginning of a heading, when headings are used in messages along with text. Headings usually state the name and location of an addressee. This code was originally called SOM, "Start Of Message."

STX (Start of Text)—Used as a marker for beginning of text and end of heading (if used). Was originally called EOA, "End Of Address."

SUB (Substitute)—Indicates a character that is to be used to take the place of a character known to be wrong.

SYN (Synchronous Idle)—Some high-speed data communications systems use synchronized clocks at the transmitter and receiver ends. During idle periods, when there are no bit patterns to enable the receiver's clock to track the transmitter's, the receiver may drift out of sync. Every transmission following an idle period is therefore replaced by three or four SYN characters. The SYN code has a bit pattern that enables the receiver not only to lock onto the transmitter's clock, but also to determine the beginning and end points of each character. SYN characters may also be

used to fill short idle periods to maintain synchronization, hence the name.

VT (Vertical Tab)—Used to tab the carriage, print head, or cursor to the next predetermined stop (usually a line).

APPENDIX C

MANAGING THE APPLE WORKSPACE

When you boot Logo and do not insert your own disk into the Apple, a set of procedures called "AIDS is loaded into the Apple "workspace." The workspace occupies part of the Apple's 64K RAM memory. The workspace is a temporary storage area, available only when the Apple is turned on. The workspace contains about 14K of the Apple's 64K memory which is "free" for storing your procedures. This 14K is divided up into "nodes," each node taking five bytes. Thus, you have about 2800 nodes available for storage of procedures and variables. The AIDS procedures buried in your workspace takes up about 400 nodes.

If you want to conserve space, here are some suggestions:

1. Erase the AIDS procedures.

If the AIDS procedures are loaded into the workspace, they can be erased by "unburying" them and erasing them. To unbury them, type UNBURY "AIDS, then type ERALL.

The reason the AIDS procedures are buried is to allow you to save any procedures you create (temporarily stored in the workspace) to disk without also saving the AIDS.

Ideally, you should initialize your own disks for storing Logo programs so that the disks do not contain the AIDS utilities.

2. Check for buried procedures.

There are times when it is useful to bury procedures, for example, graphics utilities used to create designs. When you finish running these programs, and type ERALL, it is possible that procedures are still buried, taking up workspace memory.

To check for buried procedures, type PPS. Then unbury all packages listed and erase them. Individual procedures cannot be buried. They must be "packaged" and then the package buried or unburied. (Note: A bug may exist in your version of Logo where packages buried with the same name as a procedure won't show up when using PPS.)

3. Reboot Logo.

If space is really critical, it is possible to reboot Logo in order to free up some nodes. This works because Logo saves at least one copy of all words ever used. To see how many nodes you have, type the command, RECYCLE, then NODES. RECYCLE is a command used to reclaim once used nodes (for example, used for an input to a procedure).

4. Use the command RECYCLE.

It takes quite a few nodes to actually run a procedure. There are times when you might get the error message OUT OF SPACE. One possible solution is to use the command RECYCLE to free up once used nodes, then run the procedure. If this doesn't work, you may have to reboot Logo (see above).

5. Select variable and procedure names that are known.

Since Logo keeps a copy of all words ever used (see above), Logo tries to save space by not creating identical words. Thus, if you name an input :WORD or :LIST, i.e., use names of primitives or known words, this will take less space.

6. Make use of utilities.

It is better to have several small procedures available to be used over and over again than it is to write a long procedure that does the same thing. The best example would be when you ask the user to input information several times in a program. Use the utility GETB.NUMBER or GETB.LIST several times as necessary.

7. Run the program in pieces.

For very large programs, this is the only solution. The program itself can erase, save or load procedures from the disk to the workspace and back again as desired.

8. Use short names for variables and procedures.

Try not to do this! It is considered bad programming form to sacrifice "readability" for storage space.

APPENDIX D

LOGO ERROR MESSAGES

Logo error messages are one of its outstanding features. These messages pinpoint the problem and help make debugging programs easier than most other languages.

When you get an error message, always pay close attention to it and try to figure out each part of the message. The error message generally comes in three parts as follows:

Part 1 ..... the nature of the problem
Part 2 ..... the procedure where the problem occurs
Part 3 ..... the line where the problem occurs

The form of the error message is as follows:

(error message) IN (procedure)
(restates line of code containing error)

Here are the two most common examples:

1) Giving Logo an unmarked word it doesn't know

```
TO BOX :X
FD :X RT 90
FR :X RT 90
FD :X RT 90
FD :X RT 90
END

?BOX 100
I DON'T KNOW HOW TO FR IN BOX
FR :X RT 90
```

2) Giving Logo a variable name without a value

```
TO FIGURE
REPEAT 4 [FD :X RT 90]
END
```

```
?FIGURE
:X HAS NO VALUE IN FIGURE
REPEAT 2 [FD :X RT 90]
```

Logo has 42 error messages for your convenience. Here is a complete list of the error messages.

Number	Message
1	(procedure) IS ALREADY DEFINED
2	NUMBER TOO BIG
3	(symbol) ISN'T A PROCEDURE
4	(symbol) ISN'T A WORD
5	(procedure) CAN'T BE USED IN A PROCEDURE
6	(symbol) IS A PRIMITIVE
7	CAN'T FIND LABEL (symbol)
8	CAN'T (symbol) FROM THE EDITOR
9	(symbol) IS UNDEFINED
10	(procedure) DIDN'T OUTPUT TO (symbol)
11	I'M HAVING TROUBLE WITH THE DISK
12	DISK FULL
13	CAN'T DIVIDE BY ZERO
14	END OF DATA
15	FILE ALREADY EXISTS
16	FILE LOCKED
17	FILE NOT FOUND
18	FILE IS WRONG TYPE
19	TOO FEW ITEMS IN (list)
20	NO MORE FILE BUFFERS
21	CAN'T FIND CATCH FOR (symbol)
22	(symbol) NOT FOUND
23	OUT OF SPACE
24	(procedure) CAN'T BE USED IN PROCEDURE
25	(symbol) IS NOT TRUE OR FALSE
26	PAUSING . . .
27	YOU'RE AT TOPLEVEL
28	STOPPED!
29	NOT ENOUGH INPUTS TO (procedure)
30	TOO MANY INPUTS TO (procedure)
31	TOO MUCH INSIDE PARENTHESIS
32	TOO FEW ITEMS IN (list)
33	CAN ONLY DO THAT IN A PROCEDURE
34	TURTLE OUT OF BOUNDS
35	I DON'T KNOW HOW TO (symbol)

36	(symbol) HAS NO VALUE
37	) WITHOUT (
38	I DON'T KNOW WHAT TO DO WITH (symbol)
39	DISK VOLUME MISMATCH
40	DISK IS WRITE PROTECTED
41	(procedure) DOESN'T LIKE (symbol) AS INPUT
42	(procedure) DIDN'T OUTPUT

APPENDIX E

APPLE LOGO STARTUP

Keyboard

SHIFT N = [	Left bracket (Apple II and II+)
SHIFT M =]	Right bracket (Apple II and II+)
CTRL-G	Emergency stop

Editing At The Command Level

→	Moves cursor one space to the right
←	Deletes back (left) one space
CTRL-A	Moves cursor to beginning of line
CTRL-B	Moves cursor back or left one space without erasing
CTRL-D	Deletes character under cursor
CTRL-E	Moves cursor to end of line
CTRL-K	"Kills" all characters to the right of the cursor
CTRL-Y	Restores all characters "yanked" by CTRL-K

Editing in the Editor

All of the above, plus:

CTRL-N	Moves cursor to the next line down
CTRL-O	Opens a new line at the cursor location
CTRL-P	Moves the cursor up to the previous line
CTRL-C	Exits the editor saving all changes

Creating and Editing Procedures

TO BOX	Used to define a procedure named BOX

END	Command indicating procedure definition is completed
EDIT "BOX	Directs Logo to display BOX procedure; enter edit mode
CTRL-C	Exits edit mode, saving changes

Workspace and Storage Primitives

POTS	Displays titles of all procedures in memory
PO "BOX	Displays program named BOX
POALL	Displays titles and listing for all procedures in memory and all variable values
ERASE "BOX	Erases procedure named BOX from workspace
ERALL	Erases all procedure and all variables in memory
SAVE "BOX	Saves procedure named BOX onto disk
CATALOG	Displays procedures stored on file disk
ERASEFILE "BOX	Erases file named BOX from disk
LOAD "BOX	Loads file named BOX from disk into workspace
.PRINTER 1	Activates printer, sends all screen output to printer (if control card is in slot #1)
.PRINTER 0	Deactivates printer

Turtle Graphics

FORWARD 60 or FD 60	Moves turtle 60 steps in direction it is heading
BACK 60	or BK 60Moves turtle 60 steps back
RIGHT 60	or RT 60Turns turtle 60 degrees clockwise
LEFT 60	or LT 60Turns turtle 60 degrees counterclockwise

PENUP or PU	Lifts pen so turtle can be moved without leaving tracks
PENDOWN or PD	"Normal" trail mode is restored

HIDETURTLE or HT	Turtle is hidden, but can still leave tracks
SHOWTURTLE or ST	Turtle reappears

CLEARSCREEN or CS	Erases all graphics; restores turtle at HOME (0,0) position

HOME	Directs turtle to center of screen (0,0) position
POS	Outputs turtle position as X, Y coordinate
SETX 70	Directs turtle to position X = 70
SETY 70	Directs turtle to position Y = 70

XCOR	Outputs turtle's current X coordinate position
YCOR	Outputs turtle's current Y coordinate position

PENCOLOR or PC	Displays current pencolor number
BACKGROUND or BG	Displays current background number
SETPC n	Sets pencolor tracks according to the code below
SETBG n	Sets background color according to the code below

Color Code	
0 Black	Normal
1 White	BG 6
2 Green	PC 1
3 Violet	
4 Orange	
5 Blue	
6 Black	

Screen Modes

TEXTSCREEN or CTRL-T	Normal text mode
SPLITSCREEN or CTRL-S	Normal graphics screen
FULLSCREEN or CTRL-L	Used to view full graphics screen

WRAP	"Normal" turtle graphics condition; any movement exceeding screen limits wraps around other side
WINDOW	Turtle does not WRAP; movement exceeding screen limits are not visible
FENCE	Turtle movement confined to screen

Other Primitives

PRINT input Displays input on screen. Input must be a number, word or list
REPEAT 4 [FD 50 RT 90]
This example of REPEAT draws a box

+	Addition
−	Subtraction
*	Multiplication
/	Division

APPENDIX F

SUMMARY OF LOGO FEATURES

Here is a summary of the features that contribute to Logo's simplicity:

1. **Logo is interpreted and interactive.**

   This means that Logo will try to carry out your commands right away, one line at a time. An interpreted language makes it extremely easy to explore new programming ideas. An interactive language is the major characteristic that makes BASIC easy to use. A noninteractive language like FORTRAN or PASCAL must be "compiled."

2. **Logo is modular.**

   Most modern languages today (except BASIC) share this property. This characteristic of Logo allows programming projects to be broken up into several smaller simpler parts. Each program is called a "procedure."

3. **Logo is extensible.**

   Logo has certain built-in key words like "PRINT" and "FORWARD." These key words are called "primitives," words that the computer already knows. However, you can create new procedures which are used just like the primitives. An extensible language uses the same "syntax" for both primitives and programmed procedures.

4. **Logo uses nontyped variables.**

   Most languages have several "types" of variables: numbers, strings, arrays, integers, real, etc. Logo uses words (which includes numbers) and lists of words. The same variable in Logo can be an integer in one part of the program and a character later on.

5. **Logo uses automatic dynamic memory allocation.**

   This means you don't have to worry about or declare the size of lists or words before (or while) you use them.

6. **Logo has dynamic scoping.**

   Roughly speaking this means that Logo will use local variables first, if defined. Otherwise, it will look for global definitions.

7. **Logo uses helpful error messages.**

   Logo error messages explain what caused the error and where the error occurs. Debugging programs becomes easy to do.

8. **Logo has turtle-graphics.**

   Turtle-graphics provide a marvelous way to begin learning a new language. The visual feedback is highly motivating and not only teaches the language's vocabulary and structure, but also builds a foundation for the ideas of structured programming.

Even though Logo is easy to use, it is quite powerful and open-ended for the following reasons:

1. **Logo is modular.**

   Like most languages (except BASIC), Logo uses modular procedures that give structure to programs. Complex procedures are built up from simple ones. This is the same property described above.

2. **Logo is recursive.**

   This property of Logo, which allows a procedure to call itself, makes Logo extremely powerful as seen throughout the book.

3. **Logo can use complex data objects.**

   Logo can use words, lists, lists of lists, and programs as input to procedures. Most older languages do not have this property.

Languages that have these or similar properties; that is languages that are both simple and powerful, can be summed up in one phrase. Logo is a "high-level language" that begins to provide us with the capability of our natural language, English. Low-level languages talk to machines, while high-level languages talk to people.

INDEX

A

ADDUP procedure, 238-240
Algebra, 266-270
  graphing functions, 266-268
  solving equations, 269-270
AND, 282-287
ARCR procedure, 105
Arithmetic, 248-254, 415-417
Artificial intelligence (AI), 198
ASCII
  definitions, 432-435
  table, 429-432
Assigning variable values, 231
Astroid, 257-268
AVERAGE, 272-273

B

BACK, 27
BACKGROUND, 65
BEGIN procedure, 364-365
Box chart programming method, 133-135
BURY, 114-117
Burying procedures, 114-117
BUTFIRST, 182-187
BUTLAST, 192-194

C

Cartesian coordinate system, 57-58, 71
CATALOG, 109
Character strings, 177-179
CHOOSEWORD procedure, 365-366
CIRCLER procedure, 104-106
CLEARSCREEN, 27
Color primitives, 63-65
  SET BG, 64-65
  SET PC, 64-65
  BACKGROUND, 65
  PENCOLOR, 65
Command levels, 48-49
  EDIT level, 49
  Procedure level, 49
  TO level, 48
Commands, 24-28, 78-79, 82, 409
  Screen, 420-421
  text, 420-421
  vs. operations, 79
COMPLEX procedure, 161-164
Complex recursion, 161-163
Conditionals, 274, 418-419
  AND, 283-288
  IF, 276-283
  IFFALSE, 288-289
  IFTRUE, 288-289
  NOT, 283-288
  OR, 283-288
  TEST, 288-289
Control characters, 31, 36-37, 51, 425
COUNT, 207-208
COUNT.UP procedure, 155-158
Cube, 258-259
Cursor motion keys, 43
CUTWORD procedure, 187-189

D

DASHMAKE procedure, 368-370
DEFINE, 182
Disks, 100-102, 112-114
  handling, 112-114
  initializing, 100-102
  protecting, 114
  saving a list on, 346-359
DIST.HOME procedure, 250-251
Dotted words as inputs, 83-85

E

EDIT level, 49
Editing, 28-32
  keys, 31, 36
  primitives, 426-427
END, 33
ERASE, 108-109
ERASEFILE, 110
Error messages, 439-441

EMPTYP - chap 11

Error trapping, 306-308
Escape sequence, 177-178

F

FENCE, 63
Files
  commands, 422-423
  graphing from, 340-342
  names, 111-112
FINE procedure, 372-374
FIRST, 182-187
FORWARD, 27, 94-95
FPUT, 203-207
FULLSCREEN, 52-55

G

Geometry, 254-260, 265-266
  astroid, 257-258
  cube, 258-260
GETA.NUMBER, 236
GETB.NUMBER, 293-296
GETB.YN, 308-310
GETC.NUMBER, 308-310
GET.DATA procedure, 349-352
GETGUESS procedure, 370-372
GET.INFO procedure, 330-334
GET.KEY, 296-297
Global variables, 223-227
GO.AHEAD procedure, 334
GRAPHFN procedure, 267-270
Graphics utilities, 102-106
  ARCR, 105
  CIRCLER, 104-105
Graphing from a file, 340-342

H

Haiku poetry program, 212-213
Handling disks, 112-114
Hangman program, 361-384
Hard copy, 107-108, 118-119
HARDCOPY procedure, 118-119
HOME, 58

I

IFFALSE, 288-289
IF, 276-283
  complete form with three inputs, 280-283
  simple form with two inputs, 276-280
IFTRUE, 288-289
Infix operations, 249
INIT procedure, 349-350
Initializing disks, 100-102
Inputs, 82-93
  combined with MAKE, 232
  creating variables with, 216
  giving values to, 217-220
  names, 88-89
  stopping recursion with, 170-171
  types, 82-87
  used to assign variable values, 231
  writing procedures with multiple, 89-93
Interactive graphics program, 232-234
Interactive programs, 303-306
Interest program, 169-170
ITEM, 208-209

K

Keyboard, 20-22
  Apple II and II+, 21
  Apple IIe, 22
KEYP, 296-297
Keys
  cursor motion, 43
  editing, 31, 36

L

LAST, 192-194
LEFT, 27
LISP, 198
LIST, 202-203
List inputs, 82-83, 179-190, 197-203, 412-415
LOAD, 110
Loading the disk, 22-24
LOCAL, 229-230
Local variables, 223-227

Looping, 149-150
LPUT, 203-207

M

MAKE, 216, 220-234
  combined with inputs, 232
  comparing inputs with, 230-232
  used to assign variable values, 231
  used with READLIST, 227-230
MAKEGRAPH program, 328-346
Math quiz program, 310-316, 322-323
Math utilities, 270-273
Mathematics, 245-273
  algebra, 266-270
  arithmetic, 248-254
  geometry, 254-260, 265-266
  number types, 248
  scientific notation, 247
  trigonometry, 260-266
MEMBERP, 289
Multiple inputs, 89-93
MULTUP procedure, 240

N

Naming files, 111-112
NOT, 283-288
Number inputs, 82
Number types, 247
Numerical conversion, 429-432

O

On line editing, 28-32
OOPS procedure, 374-377
Operations, 79-82
  arithmetic, 415-417
  as inputs, 85-86
  infix, 250
  logical, 419
  prefix, 250
OR, 281-286
Output, 234-242
  used in recursive procedures, 237-242
  used to create variables, 234-237

P

Packaging procedures, 114-117
PACKAGE, 114-117
PENCOLOR, 65
PENDOWN, 27-28
PENUP, 27-28, 95
PGKALL, 354
PLOT.DATA procedure, 335-336
Plotting data, 190-192
PO, 107-108
POALL, 224
PONS, 224
POS, 58-60
POTS, 107
PPROP, 392
Predicates, 277, 289-293, 297-298
Prefix operations, 248
Primitives, 33, 78-82, 423-425
  color, 63-65
  editing, 426-427
  giving formal definitions to, 93-96
  property list, 389-392
PRINT, 58-60, 68-70
.PRINTER, 118-119
Procedure level, 49
Procedures, 32-42, 78-82
  ADDUP, 238-240
  ARCR, 105
  BEGIN, 364-365
  burying, 114-117
  CHOOSEWORD, 365-366
  CIRCLER, 104-106
  COMPLEX, 161-164
  COUNT.UP, 155-158
  CUTWORD, 187-189
  DASHMAKE, 368-370
  DIST.HOME, 251-252
  erasing, 108-109
  FINE, 372-374
  GET.DATA, 349-352
  GETGUESS, 370-372
  GET.INFO, 330-334
  GET.KEY, 296-297
  GO.AHEAD, 334
  GRAPHFN, 267-270
  HARDCOPY, 118-119

Procedures—cont
INIT, 347-348
managing, 107-108
OOPS, 374-377
packaging, 114-117
PLOT.DATA, 335-336
PROCESS, 370-372
REPLACE, 222
SAVE.DATA, 353-354
SETSCENE, 367-368
SETUP.GRAPH, 334-335
SHOW.DATA, 334
SUMLIST, 241
writing with multiple inputs, 89-93
PROCESS procedure, 370-372
PRODUCT, 250-251
Programming aids, 318-320
Programming techniques, 123-146
box chart method, 133-135
top-down method, 126-144
Programs
Hangman, 361-384
interactive, 303-306
interactive graphics, 232-234
interest, 169-170
MAKEGRAPH, 328-346
Math quiz, 310-316, 322-323
Senatorial chambers, 164-167
States and capitals, 393-406
STORE.DATA, 346-359
Property list primitives, 389-392
Property lists, 389-392, 403-406
and arrays, 403-404
and data bases, 404-406
and tables, 401-403
Protecting disks, 114

R

RANPICK, 209-212
READLIST, 220, 228
used with MAKE, 227-230
Recursion, 147-174
analogies, 164
painting fences, 167-168
senatorial chambers, 164-167
complex, 161-163
Recursion—cont
simple, 149-160
using IF-Condition-STOP, 155-157
with graphics, 150-155
with text, 149-150
Stopping with an input, 170-171, 316-317
REMAINDER, 251
REMARK, 319-320
REPEAT, 39-40, 95-96
REPLACE procedure, 222
RIGHT, 27
ROUND, 95
Rules of precedence, 247

S

SAVE.DATA procedure, 353-354
Saving
a file on disk, 100-102, 109-112
a list on disk, 346-359
SC, 321-322
Scientific notation, 245
Screen commands, 420-421
Screen formatting utilities, 320-322
SC, 321-322
SP, 320
Screen modes, 49-55
FULLSCREEN, 52-55
SPLITSCREEN, 50-52
Senatorial chambers program, 164-167
SENTENCE, 199-203
SETBG, 64-65
SETPC, 64-65
SETPOS, 253
SETSCENE procedure, 367-368
SETUP.GRAPH procedure, 334-335
SETX, 59
SETY, 59
SHOW.DATA procedure, 334
SHOWTURTLE, 25, 27
Simple recursion, 149-160
using IF-condition-STOP, 155-157
with graphics, 150-155
with text, 149-150
SIN, 260-263
SP, 320

SPIRAL, 232-234
SPLITSCREEN, 50-52
Square roots, 249
SQRT, 249
States and capitals program, 393-406
STEER, 296-297
Storage, 109-111
STORE.DATA program, 346-359
Strings, character, 177-179
Structure, 75-78
SUM, 250
SUMLIST procedure, 241
System requirements, 20

T

TEST, 288-289
Text commands, 420-421
TEXTSCREEN, 66-70
TO level, 48
Trigonometry, 260-265
Top-down programming, 126-144
TOWARDS, 259
Turtle boundaries, 55-63
  FENCE, 63
  WINDOW, 60-63
  WRAP, 55-60
Turtle graphics, 24-28, 409-412

U

Utilities
  graphics, 102-106
  math, 270-273
  screen formatting, 320-322

V

Variables, 83, 415
  as inputs, 88
  assigning values to, 231
  creating, 215-217, 220-227, 234-237
  global, 223-227
  local, 223-227

W

WINDOW, 60-63
WORD, 198-199
Word inputs, 82, 177-179, 182-190, 412-415
  unmarked, 179
Workspace, 106-117
  erasing procedures in, 108-109
  managing procedures in, 107-108, 421-422, 437-438
  permanent storage, 109-111
  primitives, 117
WRAP, 55-60

X

XCOR, 59-60
XY coordinate system, 57-58, 71

Y

YCOR, 59-60

Making Computers Simple

Sams/Waite Primers

Donald Martin
Marijane Paulsen
Stephen Prata

APPLE® LOGO

Startup Card

Apple is a registered trademark of Apple Computer Inc.

© 1984
by Howard W. Sams
4300 W. 62nd Street
Indianapolis, IN 46268

Keyboard

SHIFT N = [	Left bracket (Apple II and II+)
SHIFT M =]	Right bracket (Apple II and II+)
CTRL-G	Emergency stop

Editing at the Command Level

→	Moves cursor one space to the right
←	Deletes back (left) one space
CTRL-A	Moves cursor to beginning of line
CTRL-B	Moves cursor back or left one space without erasing
CTRL-D	Deletes character under cursor
CTRL-E	Moves cursor to end of line
CTRL-K	"Kills" all characters to the right of the cursor
CTRL-Y	Restores all characters "yanked" by CTRL-K

Editing in the Editor

All of the above, plus:

CTRL-N	Moves cursor to the next line down
CTRL-O	Opens a new line at the cursor location
CTRL-P	Moves the cursor up to the previous line
CTRL-C	Exits the editor saving all changes

Creating and Editing Procedures

TO BOX	Used to define a procedure named BOX
END	Command indicating procedure definition is completed
EDIT "BOX	Directs Logo to display BOX procedure; enter edit mode
CTRL-C	Exits edit mode, saving changes

Workspace and Storage Primitives

POTS	Displays titles of all procedures in memory
PO "BOX	Displays program named BOX
POALL	Displays titles and listing for all procedures in memory and all variable values
ERASE "BOX	Erases procedure named BOX from workspace
ERALL	Erases all procedure and all variables in memory
SAVE "BOX	Saves procedure named BOX onto disk
CATALOG	Displays procedures stored on file disk
ERASEFILE "BOX	Erases file named BOX from disk
LOAD "BOX	Loads file named BOX from disk into workspace
.PRINTER 1	Activates printer, sends all screen output to printer (if control card is in slot #1)
.PRINTER 0	Deactivates printer

Turtle Graphics

FORWARD 60 or FD 60	Moves turtle 60 steps in direction it is heading
BACK 60 or BK 60	Moves turtle 60 steps back
RIGHT 60 or RT 60	Turns turtle 60 degrees clockwise
LEFT 60 or LT 60	Turns turtle 60 degrees counterclockwise
PENUP or PU	Lifts pen so turtle can be moved without leaving tracks
PENDOWN or PD	"Normal" trail mode is restored
HIDETURTLE or HT	Turtle is hidden, but can still leave tracks
SHOWTURTLE or ST	Turtle reappears
CLEARSCREEN or CS	Erases all graphics; restores turtle at HOME (0,0) position
HOME	Directs turtle to center of screen (0,0) position
POS	Outputs turtle position as X, Y coordinate
SETX 70	Directs turtle to position X = 70
SETY 70	Directs turtle to position Y = 70
XCOR	Outputs turtle's current X coordinate position
YCOR	Outputs turtle's current Y coordinate position
PENCOLOR or PC	Displays current pencolor number
BACKGROUND or BG	Displays current background number
SETPC n	Sets pencolor tracks according to the code below
SETBG n	Sets background color according to the code below

Color Code	
0 Black	Normal
1 White	BG 6
2 Green	PC 1
3 Violet	
4 Orange	
5 Blue	
6 Black	

Screen Modes

TEXTSCREEN or CTRL-T	Normal text mode
SPLITSCREEN or CTRL-S	Normal graphics screen
FULLSCREEN or CTRL-L	Used to view full graphics screen
WRAP	"Normal" turtle graphics condition; any movement exceeding screen limits wraps around other side
WINDOW	Turtle does not WRAP; movement exceeding screen limits are not visible
FENCE	Turtle movement confined to screen

Other Primitives

PRINT input Displays input on screen. Input must be a number, word or list

REPEAT 4 [FD 50 RT 90]

This example of REPEAT draws a box

+	Addition
−	Subtraction
*	Multiplication
/	Division

Need Other Books in the Sams/Waite Primer Series?

YOUR OWN COMPUTER (2nd Edition)
Shows how to choose a computer for your own uses and explores applications, buzzwords, programs, hardware, and peripherals for 30 different models.
Waite and Pardee
No. 21860 ..... **$8.95**

BASIC PROGRAMMING PRIMER (2nd Edition)
Gives you fundamental BASIC keywords, statements, and functions usable with the IBM* PC, Apple* II, or any other computer running a variation of Microsoft BASIC. Also covers advanced BASIC, new game program listings, more.
Waite and Pardee
No. 22014 ..... **$17.95**

MICROCOMPUTER PRIMER (2nd Edition)
Shows you basic computer concepts, the electronics behind the logic, what happens inside the computer as a program runs, and a little about languages and operating systems.
Waite and Pardee
No. 21653 ..... **$14.50**

COMPUTER GRAPHICS PRIMER
A classic Sams best seller that helps you learn all types of graphics programming, including animation. Many program examples written in Applesoft.
Mitchell Waite
No. 21650 ..... **$15.95**

C™ PRIMER PLUS
A clear, complete introduction to the C language that guides you in the fundamentals, covers use of Microcomputer C with assembly language, and contains many sample programs usable with any standard C compiler.
Waite, Prata, and Martin
No. 22090 ..... **$19.95**

SuperCalc* PRIMER
Makes it much easier for you to find your way out of a sticky calculation, hit the right keys to get a given answer, or otherwise get the most out of your SuperCalc spreadsheet program. Fully illustrated.
Waite, Burns, and Venit
No. 22087 ..... **$16.95**

TIMEX SINCLAIR BASIC PRIMER, WITH GRAPHICS
Easiest, most effective introduction ever to computers and programming. Graphics let you "see" your commands work as your own programs gradually develop into realities.
Waite and Chapnick
No. 22077 ..... **$9.95**

UNIX™ PRIMER PLUS
Presents the elements of UNIX clearly, simply, and accurately, for ready understanding by anyone in any field who needs to learn, use, or work with UNIX in some way. Fully illustrated.
Waite, Martin, and Prata
No. 22028 ..... **$19.95**

SOUL OF CP/M* : HOW TO USE THE HIDDEN POWER OF YOUR CP/M SYSTEM
Teaches you how to use and modify CP/M's internal features, use CP/M system calls, and more. You'll need to read *CP/M PRIMER*, or be otherwise familiar with CP/M's outer-layer utilities.
Waite and Lafore
No. 22030 ..... **$18.95**

CP/M BIBLE: THE AUTHORITATIVE REFERENCE GUIDE TO CP/M
Gives you instant, one-stop access to all CP/M keywords, commands, utilities, conventions, and more. A must for any computerist using any version of CP/M.
Waite and Angermeyer
No. 22015 ..... **$19.95**

CP/M PRIMER (2nd Edition)
Completely updated to give you the know-how to begin working with new or old CP/M versions immediately. Includes CP/M terminology, operation, capabilities, internal structure, and more.
Waite and Murtha
No. 22170 ..... **$16.95**

PASCAL PRIMER
Guides you swiftly through Pascal program structure, procedures, variables, decision making statements, and numeric functions. Contains many useful examples and eight appendices.
Waite and Fox
No. 21793 ..... **$17.95**

UNIX and C are trademarks of Bell Laboratories • CP/M is a registered trademark of Digital Research, Inc. • IBM is a registered trademark of International Business Machines Corporation • UCSD Pascal is a trademark of the Regents of the University of California, San Diego Campus • Apple is a registered trademark of Apple Computer, Inc. • SuperCalc is a registered trademark of Sorcim Corporation

SAMS BOOK & SOFTWARE ORDER CARD

Catalog No.	Qty.	Price	Total	Catalog No.	Qty.	Price	Total

Subtotal
Add local tax where applicable
Add Handling Charge 2.00
Total Amount Enclosed

☐ Check ☐ Money Order
☐ MasterCard ☐ Visa

Account Number _____ Expires _____

Name (print _____

Signature _____

Address _____

City _____ State _____ Zip _____

Offer good in USA only. Prices subject to change without notice. Full payment must accompany your order.

X0438

Call 1-800-428-3696 for the name of your nearest Sams retailer or distributor (residents in IN, AK, HI call 317-298-5566).

If your retailer or distributor doesn't stock the Sams publication you need, you can order it through him or directly from us. Orders placed directly with us are subject to a $2.00 additional handling charge per order.

PHONE ORDERS

Depending on where you live, call either number listed above and charge your Sams publication to your VISA or MASTERCARD.

MAIL ORDERS

Use the order form below, or if it's missing, send your order on a plain piece of paper.

(1) Include your name, street or RD address, city, state, and zip.

(2) Tell us the titles of the books you need, the product numbers (see other side of this card) and the quantity of each one you'd like.

(3) Add up the total cost for the books, add your state's sales tax if applicable, and then add $2.00 for handling.

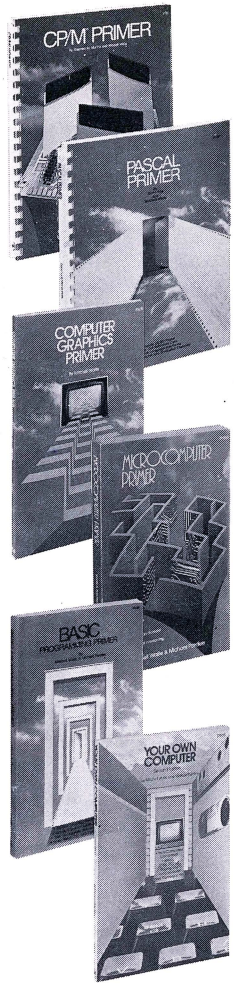
(4) Include your check or money order for the full amount due, or

(5) Charge it to your VISA or MASTER CARD. To charge it, list your account number and the card expiration date. Shipping charges will be added to credit card orders.

(6) Mail your order to:
Howard W. Sams & Co., Inc.
Dept. No. X0438
4300 West 62nd St., P.O. Box 7092
Indianapolis, Indiana 46206

In Canada, contact Lenbrook Electronics, Markham, Ontario L3R 1H2

All books available from Sams Distributors, Bookstores, and Computer Stores. Offer good in U.S. only. Note: Distributor, Computer Store and Dealer inquiries are welcome.



See other side for list of Sams/Waite Primers and their prices.

Place Stamp Here

HOWARD W. SAMS & CO., INC.

4300 WEST 62ND STREET P.O.BOX 7092

INDIANAPOLIS, INDIANA 46206

SAMS

The Waite Group

Apple® Logo Programming Primer

- Box charts illustrate program structure
- Clear diagrams of Logo syntax
- Humanized programming techniques
- Quality programming through utilities
- Special focus on recursion

The Logo programming language is sweeping the schools of America. Its structured approach to programming offers the beginner a clear and concise way to learn the most important skill of our times. One feature of the Logo language which appeals to young and old students alike is the "Turtle Graphics" system. With a series of simple commands, the turtle can be used to make many creative graphic designs.

This book provides a format for learning this new language. The various elements of Logo are presented in the easy to read style that characterizes the Waite Group books. The book can be used for private or group instruction. Each chapter includes practice exercises that will help to reinforce your learning of the concepts presented therein.

The complex topic of recursion is broken down and presented in a series of simple examples. Several cases of recursion are discussed in detail, with an abundance of graphic analogies to support the text. Logo primitives are organized into logical groups and presented at a comfortable pace with many examples. One chapter is totally dedicated to humanizing your programs so they are more user friendly. Two complete computer games, "Hangman" and "States and Capitals," are developed in detail, providing real examples of this exciting new language.

Howard W. Sams & Co., Inc.
4300 West 62nd Street, Indianapolis, Indiana 46268 U.S.A.

$19.95/22342

ISBN: 0-672-22342-2

SAMS™ **Apple® Logo Programming Primer** Martin, Prata & Paulsen 22342